Escuela
Politécnica
Superior

Representación gráfica de un océano utilizando técnicas de iluminación y renderizado 3D

Grado en Ingeniería Multimedia

Trabajo Fin de Grado

Autor:
Antonio José Martínez García
Tutor/es:
Rafael Molina Carmona

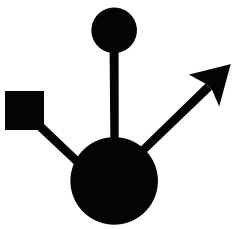
Universitat d'Alacant
Universidad de Alicante

Julio 2021

Representación gráfica de un océano utilizando técnicas de iluminación y renderizado 3D

Autor
Antonio José Martínez García

Tutor/es
Rafael Molina Carmona
Ciencia de la Computación e Inteligencia Artificial



Grado en Ingeniería Multimedia

Escuela
Politécnica
Superior

Universitat d'Alacant
Universidad de Alicante

ALICANTE, Julio 2021

Resumen

El proyecto se centra en el estudio y desarrollo de las técnicas gráficas necesarias para la representación de un océano en un videojuego. Para ello, se hace uso de un motor gráfico desarrollado previamente por el usuario al que se va a incorporar esta funcionalidad.

Al comienzo, se ha dividido la funcionalidad en las secciones de oleaje, rendimiento e iluminación, realizando una investigación de cada una de ellas. La de oleaje presenta los algoritmos con los que es posible animar el océano, la de optimización presenta los métodos que se suelen utilizar para mejorar el rendimiento de la aplicación reduciendo el número de vértices o haciendo más eficiente el cálculo de estos algoritmos, y la de iluminación muestra las técnicas con las que se obtiene un mejor resultado visual.

Tras esto, en cada sección se ha realizado una elección de los métodos más próximos a la idea de representar un océano y se han desarrollado después de diseñar la estructura base del motor gráfico sobre la que se ha trabajado. Se destacan los algoritmos de Ondas compuestas, Gerstner y Tessendorf que han sido los escogidos en la sección de oleaje. En la de iluminación, se remarcan algunas técnicas importantes de iluminación del agua como los reflejos y refracciones.

Finalmente, se ha incorporado una interfaz que permite modificar y experimentar en tiempo real con las propiedades de las funcionalidades desarrolladas, facilitando el análisis de los resultados al mostrar estadísticas del rendimiento de la aplicación. Con estas estadísticas se han realizado comparaciones de las diferentes técnicas implementadas con las que se han extraído los resultados y conclusiones del proyecto.

Concluyendo, los algoritmos de Ondas compuestas y Gerstner son más fáciles de paralelizar moviéndolos a la gráfica que el de Tessendorf, obteniendo una mejora de rendimiento considerable. Mientras, las técnicas de iluminación que han aportado un mayor realismo visual han sido el mapeado de normales, los reflejos y refracciones.

Abstract

The project focuses on the study and development of the graphic techniques necessary for the representation of an ocean in a video game. To do this, use is made of a graphic engine previously developed by the user to which this functionality is to be incorporated.

At the beginning, the functionality has been divided into the sections of waves, optimization and lighting, and an investigation of each of them has been carried out. The wave section presents the algorithms with which it is possible to animate the ocean, the optimization section presents the methods that are usually used to improve the performance of the application by reducing the number of vertices or making the calculation of these algorithms more efficient, and the lighting section shows the techniques with which a better visual result is obtained.

After this, in each section a choice of the methods closest to the idea of representing an ocean has been made and developed after designing the base structure of the graphics engine on which the work has been carried out. The Composite Waves, Gerstner and Tessendorf algorithms are the ones chosen in the wave section. In the lighting section, some important water lighting techniques such as reflections and refractions are highlighted.

Finally, an interface has been incorporated that allows the user to modify and experiment in real time with the properties of the developed functionalities, facilitating the analysis of the results by displaying statistics of the application's performance. With these statistics, comparisons have been made of the different techniques implemented with which the results and conclusions of the project have been drawn.

In conclusion, the Compound Waves and Gerstner algorithms are easier to parallelise by moving them to the graph than the Tessendorf algorithm, resulting in a considerable performance improvement. Meanwhile, the illumination techniques that have provided greater visual realism have been normal mapping, reflections and refractions.

Resum

El projecte se centra en l'estudi i desenvolupament de les tècniques gràfiques necessàries per a la representació d'un oceà en un videojoc. Per a això, es fa ús d'un motor gràfic desenvolupat prèviament per l'usuari al qual s'incorporarà aquesta funcionalitat.

Al començament, s'ha dividit la funcionalitat en les seccions d'onatge, rendiment i il · luminació, realitzant una investigació de cadascuna d'elles. La d'onatge presenta els algorismes amb els quals és possible animar l'oceà, la d'optimització presenta els mètodes que se solen utilitzar per a millorar el rendiment de l'aplicació reduint el nombre de vèrtexs o fent més eficient el càlcul d'aquests algorismes, i la d'il · luminació mostra les tècniques amb les quals s'obté un millor resultat visual.

Després d'això, en cada secció s'ha realitzat una elecció dels mètodes més pròxims a la idea de representar un oceà i s'han desenvolupat després de dissenyar l'estructura base del motor gràfic sobre la qual s'ha treballat. Es destaquen els algorismes d'Ones compostes, Gerstner i Tessendorf que han sigut els triats en la secció d'onatge. En la d'il · luminació, es remarquen algunes tècniques importants d'il · luminació de l'aigua com els reflexos i refraccions.

Finalment, s'ha incorporat una interfície que permet modificar i experimentar en temps real amb les propietats de les funcionalitats desenvolupades, facilitant l'anàlisi dels resultats en mostrar estadístiques del rendiment de l'aplicació. Amb aquestes estadístiques s'han realitzat comparacions de les diferents tècniques implementades amb les quals s'han extret els resultats i conclusions del projecte.

Concloent, els algorismes d'Ones compostes i Gerstner són més fàcils de paral · lelitzar movent-los a la gràfica que el de Tessendorf, obtenint una millora de rendiment considerable. Mentrestant, les tècniques d'il · luminació que han aportat un major realisme visual han sigut el mapatge de normals, els reflexos i refraccions.

Motivación, justificación y objetivo general

Desde el comienzo he tenido claro que no iba a escoger ninguno de los trabajos propuestos por los profesores, ya que ninguno acababa por convencerme. Prefería optar por la posibilidad de proponer una actividad que me encantase a la hora de desarrollarla, por lo que comencé realizando una lista con propuestas muy diferenciadas sobre los distintos campos abordados en la carrera. Tras reducir esta lista a una decena de propuestas, me di cuenta, de que contenía una gran cantidad de proyectos personales que o eran demasiado ambiciosos o no me acababan de convencer.

Un día vino a mi mente la idea de aprender a desarrollar un océano. Me gustó hasta el punto de que tras pensarlo varias veces detenidamente y consultar su viabilidad como trabajo final de grado, decidí descartar el resto de las propuestas que tenía más encaminadas ya consultadas con otros profesores. Además, me motiva adicionalmente que no se haya abordado una propuesta similar en la titulación.

Siempre me ha llamado mucho la atención y me ha parecido una especie de "magia" el cómo implementar elementos que no son sólidos en los videojuegos de tres dimensiones (3D), como puede ser el fuego, el oleaje, la luz o las partículas. Ya que, a pesar de no disponer de una gran experiencia en el desarrollo de videojuegos, conozco que cualquier elemento "sólido" puede representarse fácilmente con un modelado 3D. Por lo que me fascina el simple hecho de conocer los métodos que se utilizan en los videojuegos para simular océanos, además, siendo eficientes computacionalmente.

Se que es inviable desarrollar completamente un océano que se encuentre a la altura de la generación actual de juegos debido al tiempo reducido del que dispongo para la realización de este proyecto. Por lo cual he debido escoger entre focalizarme en una de las diversas etapas en las que se divide, o realizar un estudio de todas estas partes sin profundizar demasiado en ellas. He optado por la segunda opción, ya que como he comentado anteriormente no me gustaría estancarme en un solo procedimiento, sino que prefiero estudiar y abordar el desarrollo de todo el proceso. De hecho, desde un comienzo tras hablarlo en profundidad con el tutor no sabía si este enfoque iba a generar demasiada carga de trabajo. Pero, tras una investigación inicial, vi que era completamente viable. Sin embargo, comprendí que iba a requerir de una gran dedicación para conseguir buenos resultados.

Este trabajo se encuentra bastante relacionado con la asignatura de gráficos por computador, ya que se centra en añadir una funcionalidad adicional al motor gráfico desarrollado durante el último año de la titulación para esta asignatura. Además, la buena estructura con la que se desarrolló el motor junto a mis compañeros de equipo me permite incluir estas funcionalidades sin tener que modificar gran parte de su estructura. También, se encuentra relacionada con la asignatura de señales y sistemas y sus derivadas. Debido a que se va a tener que trabajar con ecuaciones de las ondas y aplicarles procedimientos ya vistos en la asignatura o bastante relacionados con esta.

Y por supuesto, se van a demostrar las habilidades de autoaprendizaje, investigación, pla-

nificación y desarrollo entre otras que he ido adquiriendo durante la carrera.

Agradecimientos

En primer lugar, quiero agradecer a mis amigos de la titulación que me han acompañado durante los últimos años, han hecho de este periodo una experiencia increíble en la que he podido aprender y disfrutar.

También, agradezco a mi novia y amigos por haberme apoyado y motivado en el desarrollo del proyecto, aportándome las energías necesarias y la motivación adicional que me faltaba en algunas ocasiones.

A mi tutor del proyecto, Rafael Molina Carmona, por resolver todas las dudas que le he planteado y aconsejarme cuáles eran las mejores decisiones que debía tomar para mejorar el resultado final del proyecto.

Sin olvidar, a mi familia que me ha estado apoyando en mis decisiones personales y aguantando desde que comencé la titulación.

Muchas gracias a todos.

Índice general

Índice de figuras

Índice de tablas

Índice de Códigos

1. Introducción

Desde el comienzo, los videojuegos siempre han buscado una semejanza lo más cercana posible a la realidad en el apartado visual con la finalidad de sumergir al usuario en experiencias más inmersivas. Por esto se han desarrollado una gran cantidad de técnicas para representar objetos tridimensionales en nuestros monitores a un coste computacionalmente asequible. Estas nos han permitido mostrarlos con una calidad superior a la que realmente tienen, siendo capaces de ser procesados en tiempo real, sin tener que esperar algunos segundos para poder visualizarlos en pantalla.

Tras ser capaces de representar objetos 3D, nos damos cuenta de que los objetos que no disponen de movilidad propia, como pueden ser una mesa o un edificio, tienen el mismo comportamiento en la realidad que como se muestran. Sin embargo, los elementos como los océanos o árboles afectados por el viento no pueden permanecer estáticos, ya que no es un comportamiento natural. Por lo que tenemos que dotarlos de movimiento, es decir, animarlos.

Los objetos en los videojuegos están conformados por una malla que se encuentra compuesta por una gran cantidad de vértices. La animación de estas mallas suele implementarse como una sucesión de fotogramas en las que se desplazan determinados vértices de un objeto a diferentes posiciones. Estas animaciones se generan en programas externos de modelado 3D que facilitan el cálculo de estas posiciones y posteriormente se exportan e incluyen en los videojuegos. Por ejemplo, la animación de saltar de un personaje se genera previamente en uno de estos programas externos y luego se incluye, por lo que siempre va a realizar el salto de la misma forma.

No es un problema que el jugador se encuentre saltando todo el rato de la misma manera, ya que se puede relacionar con la forma que el personaje ha aprendido a saltar. Sin embargo, un océano tiene una gran cantidad de posiciones distintas y es demasiado trabajo tener que calcular todas estas posiciones para grandes superficies de agua. Además, de que sería necesario un gran espacio de almacenamiento para guardarlas todas.

Una de las soluciones podría ser el cálculo de unos pocos de estos movimientos en un tamaño reducido y repetirlos en bucle para toda la superficie. Esto valdría para una pequeña superficie de agua, pero para una mayor superficie en la que el jugador se encuentre más tiempo le resultaría demasiado repetitivo y antinatural. Además, de que no se podría controlar el estado del mar dependiendo de las situaciones diferentes en las que se encuentre el jugador. Como por ejemplo, la dirección en la que se mueve el oleaje o la bravura con la que crecen las olas.

Por lo que la única opción que queda es que estos cálculos de las posiciones en las que se encuentra el mar se realicen en tiempo real y no previamente.

Ahora bien, para obtener las posiciones de las olas en cada momento hay una gran cantidad de ecuaciones físicas y matemáticas complejas que simulan un estado bastante preciso (Salmon, 2021) en el que influyen una gran cantidad de factores como la densidad del agua o las corrientes oceánicas entre otros. Estas operaciones pueden ser aplicadas y se aplican en

películas, pero la elaboración de estos cálculos para pasar simplemente de una posición a la siguiente puede llegar a llevar varios minutos.

A grandes rasgos, para que un videojuego se mueva a una velocidad aceptable de 30 fotogramas por segundo (del inglés *FPS*), el tiempo que tiene para realizar cada iteración de cálculo de operaciones y dibujado en pantalla es de unos 0,033 segundos. Y obviamente solo una fracción de ese tiempo se dedica al cálculo del oleaje ya que hay otros aspectos que también requieren ser procesados. Por lo que la aplicación de estas ecuaciones es totalmente inviable en videojuegos debido al tiempo de cómputo. Y sin entrar a profundizar en aplicaciones de realidad virtual donde se busca una tasa mínima de 60 o 120 *FPS* para evitar marear al usuario.

Durante bastante tiempo muchos investigadores han tratado de simplificar estas ecuaciones realistas o generar algunas más simples con comportamientos similares para que pudiesen ser calculadas en cortos periodos de tiempo. Eliminando muchos cálculos y procedimientos complejos para conseguir un equilibrio entre realismo y eficiencia. Este es uno de los puntos en los que se va a centrar este trabajo, en el que se va a tratar de escoger varias de estas investigaciones, implementarlas, y realizar un análisis comparativo del rendimiento y el resultado que generan. Hay algunos procedimientos que son derivados de otros en las que solo se incluyen algunos cálculos para obtener un resultado deseado ligeramente diferente del original.

Otro de los objetivos es investigar y proponer algunas de las técnicas utilizadas en videojuegos para reducir el coste de procesamiento de las mallas. Siempre son preferibles estos métodos imperceptibles para el usuario a tener que reducir la complejidad de estas ecuaciones disminuyendo la calidad visual y obteniendo una peor experiencia para el jugador. Principalmente estas técnicas se suelen basar en la reducción o eliminación de los vértices de las mallas; Porque, aunque no visualices un objeto en pantalla si no se aplican los métodos adecuados puede que se siga calculando y consumiendo tiempo de procesamiento.

Por último, no es suficiente con que el rendimiento sea perfecto y la forma que tomen las olas sean buenas si no disponemos de un aspecto visual fascinante. Por lo que se va a partir de un motor gráfico propio para implementar nuevas técnicas que mejoren el apartado visual del agua. Se va a realizar un apartado introductorio del motor explicando los elementos más relevantes que se utilizan y un vistazo a su funcionamiento interno.

Hay una gran cantidad de estéticas visuales diferentes, desde más realistas hasta más próximas a un estilo de dibujo animado o de cómic. Este apartado visual es totalmente independiente de la generación del oleaje, pero sí que tiene una repercusión en el rendimiento del juego. En este caso, se va a tratar de conseguir una estética lo más realista posible. Este realismo visual implica normalmente un mayor tiempo de cómputo, ya que los cálculos que hay que realizar suelen ser más complejos. Algunas de las técnicas visuales que vamos a abordar van a ser el reflejo y la refracción del agua, el rebote de los rayos del sol en la superficie, y los elementos cáusticos[1] cuando el agua tiene poca profundidad, además de abordar e implementar algunos de los trucos que se utilizan para simular un comportamiento realista con un coste muy bajo.

Y para finalizar se realizará un análisis para comprobar cuáles de estas técnicas de iluminación tienen un mayor impacto en el rendimiento y qué otros métodos se pueden utilizar

[1] En óptica, una **cáustica** es la envolvente de los rayos de luz reflejados o refractados por una superficie curva u objeto, o la proyección de esa envolvente de rayos en otra superficie (`https://es.wikipedia.org/wiki/Cáustica_(óptica)`).

para tratar de evitar usarlas si son demasiado costosas.

2. Metodología y planificación

Para el desarrollo del trabajo fin de grado (TFG) se va a utilizar una metodología ágil. Estas se caracterizan por ser iterativas, realizando evaluaciones del estado del proyecto cada una o varias semanas. Estas iteraciones van a facilitar la prevención de posibles riesgos y retrasos, así como conseguir una mejora en el producto final al entregar funcionalidades completamente finalizadas en cada uno de estos periodos. Además, se van a realizar reuniones con el tutor tras finalizar cada iteración para mostrarle los avances desarrollados durante esta.

En concreto, la metodología ágil que se va a emplear va a ser Kanban, va a permitir realizar periodos de iteración de dos semanas de duración. En cada uno se van a ordenar las tareas propuestas por prioridad y luego se van a especificar desglosándose en subtareas que no lleven un tiempo superior a las diez horas. Tras esto, se escogerá una de las tareas especificadas para implementarse y luego depurarse en caso de ser necesario. Una de las características de Kanban es que limita el número de tareas simultáneas que se están procesando en cada una de estas etapas. Esto va a permitir que se evite trabajar simultáneamente en demasiadas características.

Kanban está orientada a utilizarse en grupos de trabajo, por lo que se va a modificar para adaptarla a una sola persona. Para ello, se va a eliminar la columna de validación al no disponer de otra persona que valide las funcionalidades finalizadas. También, se van a limitar la implementación y la especificación a una tarea simultánea.

2.1. Tecnologías

- La herramienta que va a facilitar trabajar con este tipo de metodologías va a ser **Trello**. Es una página web que dispone de una interfaz sencilla donde se pueden crear apartados y tarjetas de tareas ordenándolas y etiquetándolas como se prefiera, como se puede ver en la figura 2.1.

- **Toggl** va a permitir llevar un control del tiempo dedicado a cada tarea realizando registros del tiempo para luego poder visualizarlos gráficamente en periodos temporales de semanas, meses o años. De esta forma se puede controlar si una semana se ha trabajado menos y tratar de recuperar esas horas en la siguiente.

- **Git** se va a emplear como la herramienta de control de versiones. Para ello se va a utilizar **GitKraken** como interfaz gráfica en Windows.

- **Microsoft Visual Studio** va a ser el entorno de desarrollo integrado que se va a utilizar para desarrollar la aplicación. Ya que es compatible con C++, el lenguaje de programación que se va a utilizar. Además, facilita la incorporación de librerías al proyecto y su compilación para Windows. También, dispone de un depurador que va a permitir analizar y encontrar fallos en la aplicación.

Figura 2.1: Captura de Trello durante las primeras etapas del proyecto

- Las especificaciones del ordenador en el que se va a desarrollar la aplicación y realizar los análisis van a ser: como gráfica una Nvidia 1050 Ti, como procesador un Intel i7 de séptima generación, 8 *GB* de memoria *RAM* y Windows 10 de 64 bits como sistema operativo.

2.2. Etapas del proyecto

El proyecto se va a dividir en las siguientes etapas:

- **Etapa 0** (1 mes): Se va a comenzar a preparar el entorno de trabajo en Microsoft Visual Studio (MVS) llevando el motor gráfico a este editor. También, se realizará una planificación inicial del proyecto.
- **Etapa 1** (2 meses): Periodo de investigación y análisis. Como el proyecto se basa en la investigación, a esta se le va a dedicar gran parte del tiempo. Al finalizar se realizará una planificación detallada del periodo restante.
- **Etapa 2** (3 meses): Implementación y corrección de errores.
- **Etapa 3** (2 meses): Realización de la memoria. Desde el inicio del proyecto se irá documentando pero será en esta etapa donde se amplie y se unifique toda esta información.

2.3. Gestión de riesgos

Este proyecto se enfoca a la investigación y no se va a lanzar a un mercado, por lo que los posibles riesgos son pocos. Sin embargo, cabe destacar alguno de los más importantes que pueden llegar a suceder.

- **Contratación laboral:** Me encuentro cursando prácticas de empresa y cabe la probabilidad que haga un contrato al finalizarlas, por lo que dispondría de menos tiempo del planificado para la realización del TFG. En caso de suceder se replanificaría la parte restante del proyecto y se pospondría la fecha de defensa si fuera necesario.

- **Pérdida de código o archivos:** Para prevenirlo, además de estar utilizando un sistema de control de versiones, se realizarán copias de seguridad periódicamente que se guardarán en distintos sistemas de almacenamiento web.
- **Enfermedad:** Se volverían a planificar las tareas y se pospondría la fecha de defensa si fuera necesario.

2.4. Producto mínimo viable

En cuanto al producto mínimo deberá disponer de al menos la generación de la malla del océano para que se pueda animar. Un algoritmo de oleaje que nos permita realizar esta animación. Y las técnicas de iluminación mínimas para que de la sensación de que sea agua, con una calidad razonablemente realista. Además, de una interfaz que permita visualizar al menos los *FPS* para poder realizar un análisis de rendimiento.

3. Estado de la cuestión

En este apartado se estudian las principales técnicas y métodos para renderizar océanos de manera realista, con el objetivo de seleccionar aquellos que se van a desarrollar posteriormente en este trabajo.

3.1. Algoritmos de oleaje

Se pueden clasificar en dos grupos de algoritmos:

- Los que se centran en la modificación de los vértices de la malla. Empleados para consumir poco rendimiento con un buen resultado, evitando reducir las mallas por debajo de un mínimo número de vértices para que la animación no se vea abrupta. Pueden calcularse en el dominio espacial o espectral.
- Los que utilizan el cálculo de las propiedades físicas del agua y se aplican sobre partículas. Empleados para obtener resultados bastante realistas a un alto coste computacional.

Se van a seleccionar algunos de estos algoritmos e investigarlos:

3.1.1. Ondas compuestas

El algoritmo de Ondas compuestas pertenece al grupo basado en modificar los vértices de una malla en el dominio espacial. Una onda plana es una onda que se desplaza en una dirección donde cada cierto tiempo sus valores de amplitud se repiten. Modificando las propiedades de la onda definidas en su ecuación se varia su comportamiento (Elmore y Heald, 1985, Cap.1.3).

Valores modificables:

- La amplitud define la altura máxima y mínima que alcanzará la ola.
- La longitud de onda define cuál es la distancia entre dos puntos iguales de la onda. Por ejemplo, en un océano calmado esta distancia será mucho mayor que en uno agitado.
- Con la frecuencia angular se puede modificar la velocidad de desplazamiento de la onda. Haciendo que el oleaje se mueva a una mayor o menor velocidad.

Modificando un poco la ecuación puede ser aplicada a una superficie de dos dimensiones, siendo capaces de variar su dirección de desplazamiento. Por lo que realizando este cálculo se obtiene como resultado la altura actual de la ola en una determinada posición X y Z, que se utiliza como tercera dimensión. Con esto, si se proporciona un plano con muchos vértices se puede calcular la altura en cada uno dependiendo la posición en la que se encuentren.

Para evitar que el movimiento sea demasiado repetitivo, se generan otras ondas simples con valores diferentes y se combinan obteniendo una onda compuesta (ver Figura 3.1). Utilizando

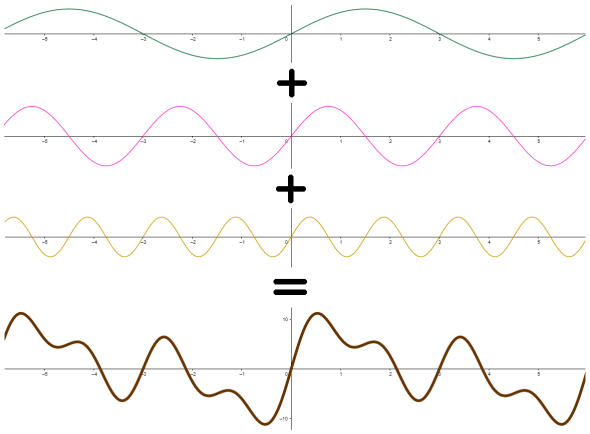


Figura 3.1: Ejemplo de la creación de una onda compuesta
Fuente: Federico Balmaceda (2015)

unos valores adecuados para cada onda se puede llegar a obtener una correcta simulación de un océano calmado, siendo más difícil apreciar esta repetición.

El coste que tiene aplicar este método suele ser bajo, ya que no implica muchas operaciones matemáticas. Sin embargo, con el uso de este algoritmo solo es posible representar océanos calmados.

Esta primera aproximación al oleaje utiliza las ecuaciones de las ondas que se han dado en diversas asignaturas durante la titulación, pueden verse desarrolladas en el documento Elmore y Heald (1985). Sirven como base para muchos de los algoritmos que se basan en la animación de mallas.

3.1.2. Gerstner

En el algoritmo anterior solo era posible calcular el desplazamiento en la altura y no en el resto de los ejes. El algoritmo de Gerstner (Stuhlmeier, 2015) parte del anterior para obtener un resultado más realista incorporando un desplazamiento en los ejes X y Z. Por lo que, también pertenece al grupo de algoritmos que se centran en la modificación de los vértices de una malla en el dominio espacial.

Este comportamiento añadido se conoce como onda de Gerstner u onda Trochoidal, siendo la idea principal que los vértices de las olas se muevan siguiendo una circunferencia. Dependiendo de lo desplazadas entre si que se encuentren estas se definirán lo agudas que van a ser las crestas de las olas. Modificando su radio se obtendría el mismo resultado como se puede apreciar en la imagen 3.2.

Además, este factor de desplazamiento puede ser modificado en tiempo real, obteniendo un océano más revuelto si se disminuye la anchura de la parte superior de las olas, o aumentándola en caso de querer obtener un océano más calmado.

Otra mejora es que, en Gerstner, la velocidad con la que se desplaza el oleaje depende de la gravedad y la longitud de la onda. Cuanto mayor sea la longitud de onda más rápido se desplazará, buscando con esto que la anchura de la cresta de la ola sea solo modificable

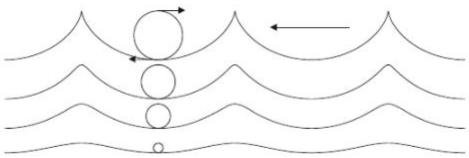


Figura 3.2: Ejemplo de onda Trochoidal
Fuente: Stuhlmeier (2015)

mediante el factor de desplazamiento; si no, la modificación de la longitud de onda afectaría a esta anchura.

3.1.3. Tessendorf

La implementación de Tessendorf (2001) presenta un modelo de oleaje estático, es decir, un modelo donde se descomponen los campos de altura de olas en una suma de ondas más simples. También, pertenece al grupo que se centra en modificar los vértices de la malla, esta vez pertenece al dominio espectral al realizar cálculos con el espectro de la onda.

Los campos de alturas son mapas donde se almacena la altura resultante de una onda en dominio temporal calculada mediante un conjunto de frecuencias en dominio espectral. Para realizar este cálculo se aplica la ecuación conocida como **Transformada de Fourier** que permite descomponer una onda en un conjunto de frecuencias. O la que se utiliza en este algoritmo, la **Transformada de Fourier Inversa**, que permite obtener una onda en dominio espacial a raíz de un conjunto de frecuencias.

Para calcular las propiedades de estas frecuencias se emplea, en primer lugar, la función de dispersión que permite relacionar la longitud de onda y la frecuencia con su gravedad de una forma similar a como se relacionaba en Gerstner. Y luego, para calcular las propiedades de las amplitudes, en el dominio de la frecuencia, se generan números aleatorios gaussianos, ya que se han realizado estudios concretando que siguen valores de altura similares a los de un océano real afectado por el viento. Estos valores son en realidad pseudo aleatorios, ya que se regulan con la función del espectro de Philips que permite modificar el valor resultante de la altura en caso de que se desplacen de forma transversal al viento y puedan generar un resultado extraño visualmente. Además, hay otras formas de generación de números aleatorios alternativas a Gauss que consiguen resultados similares.

La Transformada de Fourier Inversa permite generar un mapa de alturas discretizado (Figura 3.3) a partir de los valores aleatorios calculados con Gauss. Lo bueno, que estos campos son bastante grandes y suelen evitar que sea apreciable un efecto de repetición de patrones cuando se observan las olas. Aun así, se puede modificar el tamaño de la cuadrícula de estos campos para mejorar el resultado incrementando el coste computacional.

Además, el cálculo de los valores de las alturas en cada punto no depende del estado anterior, por lo que se necesitan menos operaciones. El problema de este algoritmo es que no se pueden realizar efectos como perturbaciones en el agua que se desplacen por la superficie del océano. Se propone el uso de una máscara para poder solventar este problema, pero la representación de la perturbación resultante es muy poco realista.

Por último, se incorporan ecuaciones para calcular el desplazamiento de las olas en los

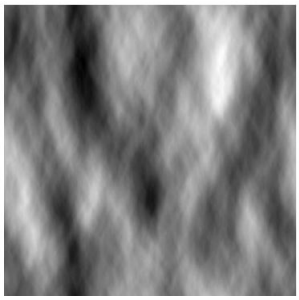


Figura 3.3: Mapa de alturas dependiente de la dirección del oleaje
Fuente: Tessendorf (2001)

vértices similares a las de Gerstner, permitiendo generar oleaje más embravecido. Incluyendo la opción de modificar las ecuaciones de dispersión para que la profundidad del agua influya en el resultado de la ola.

3.1.4. Sistemas de partículas

Los sistemas basados en la deformación de la malla que hacen uso de las Transformadas de Fourier proporcionan buenos resultados, pero en ocasiones se quieren obtener sistemas que simulen la realidad con gran precisión. Para esto se utilizan los sistemas de partículas que hacen uso de las fórmulas hidrodinámicas, centradas en el estudio de las propiedades físicas reales que dotan al agua de movimiento. Estos algoritmos son bastante costosos computacionalmente al necesitar una gran cantidad de cálculos. Por lo que usualmente dan problemas de rendimiento al ejecutarse en tiempo real en ordenadores domésticos cuando se abarcan grandes volúmenes de agua.

Uno de los sistemas más famosos es el de **Navier-Stokes** (Li y Wu, 2007, Cap.3.4), forma parte del grupo de algoritmos que utilizan el cálculo de las propiedades físicas del agua. Ya que, sus ecuaciones están basadas en la segunda ley de Newton y haciendo pequeñas modificaciones son aplicables para el cálculo de fluidos, niebla o fuego entre otros. Estas ecuaciones son muy costosas computacionalmente, solo se pueden resolver para ciertas situaciones concretas obteniendo un resultado aproximado. Sin embargo, este resultado sigue siendo mejor del que se obtiene con algoritmos que emplean las Transformadas de Fourier haciendo cálculos en el dominio espectral.

Estas ecuaciones dividen el entorno en volúmenes calculando el momento que tiene cada uno. Este cálculo se realiza sumando propiedades como la viscosidad, cambios en la presión, gravedad y otras fuerzas. En el documento Fernando (2004, Cap.38) se puede ver una explicación extendida de cada una de estas propiedades.

Tras esto, hay varios métodos que se utilizan para pasar de los resultados obtenidos en las ecuaciones a una representación del fluido. Los básicos son el método **lagrangiano** que aplica estas ecuaciones a partículas, y el método **euleriano** que las aplica a una cuadrícula (del inglés *grid*). Por ejemplo, en Müller y cols. (2003) se presenta un modelo de partículas basado en el método euleriano. Sin embargo, hay otros métodos como el **Mac Grid**, explicado en profundidad en Li y Wu (2007, Cap.3.5), que hace una combinación de los dos anteriores aplicando los resultados del desplazamiento a una malla para luego transformarlos y tras una

serie de cálculos aplicarlos a partículas.

3.1.5. Corrientes del agua en ríos

En Yu y cols. (2009) se muestra un método que permite representar el agua fluyendo en movimiento en los ríos. Para esto pueden emplearse una serie de datos del río extraídos de aplicaciones como Google Earth que utilizará el algoritmo como valores de entrada. Estos datos abordan propiedades del terreno como la dirección en la que se desplaza el agua o la anchura del canal. También, se agrega como entrada una textura que contiene el mapa de alturas calculado utilizando las Transformadas de Fourier tratadas en la explicación de Tessendorf, pero esta vez empleando un ruido de Perlin en vez de uno de Gauss. Hace uso de técnicas de los dos grupos de algoritmos descritos, pero se basa más en el de modificación de los vértices de la malla en el dominio espectral.

Lo primero que se hace es dividir el río en una cuadrícula y se reparten uniformemente partículas por esta para luego calcular la velocidad de cada partícula utilizando los parámetros de entrada introducidos, como puede ser el caudal. Estos datos del estado del río permiten calcular la velocidad del agua en cualquier posición posible pero en caso de que los datos de entrada no proporcionen información suficiente, como la del caudal, se calcularán utilizando los datos de la geometría del río y sus conexiones. Una vez que se tienen los datos listos para calcular la velocidad del agua en cualquier punto se aplicarán a cada partícula obteniendo su posición resultante.

A continuación, en cada partícula se posiciona un *sprite* con un fragmento de la textura del mapa de alturas. Para evitar que estas texturas se solapen o se dejen huecos, si las partículas están muy alejadas, se van a ir añadiendo y eliminando según sea necesario. Además, se varía el tamaño del *sprite* dependiendo de la distancia a la cámara, como se observa en la figura 3.4, para obtener una mejor resolución cuando sea necesario.

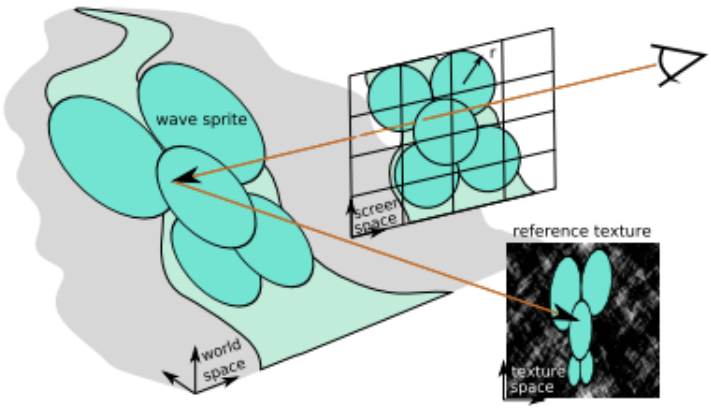


Figura 3.4: Asignación de la textura al *sprite* y cambio de tamaño del *sprite* según la distancia a la cámara

Fuente: Yu y cols. (2009)

Por último, al disponer de la malla del río se modifican las posiciones de los vértices utilizando las texturas resultantes como mapa de alturas, como se ve en Tessendorf (2001)

explicado en el apartado 3.1.3, que se obtienen tras combinar todos los *sprites*.

3.2. Técnicas de iluminación

Llegados a este punto ya se dispone de una malla o partículas de agua animadas y solo falta dotar al agua de color iluminándola correctamente para que parezca natural. Se van a investigar una serie de técnicas con las que conseguir este efecto.

3.2.1. Propiedades físicas del agua

Una de las primeras características que viene a la mente al pensar acerca del agua es la capacidad de verse reflejados en ella, por esto es la primera propiedad que se va a tratar.

Para ello, primero es necesario conocer cuál es el proceso de ***renderizado*** de una escena. Sin concretar demasiado, es el proceso en el que se envian todos los datos de los objetos de la escena a la gráfica para que genere una imagen y se dibuje posteriormente en pantalla. Es un proceso bastante costoso que suele llevar más tiempo que calcular toda la lógica del juego, realizado una única vez por ciclo.

A continuación, se detallan las propiedades recogidas en Johanson (2004, Cap.3.4) y Möller (2018, Cap.14.5):

Reflejos y refracciones: Para implementarlos es necesario *renderizar* la escena dos veces adicionales previamente y en vez de mostrar el resultado en pantalla se almacenan en una textura. Esto tiene un gran coste y por ello muchos videojuegos tratan de tener el menor número de superficies reflectantes posibles. Para evitar que el coste sea tan alto y que se realicen menos cálculos, en lugar de hacer que se *renderice* a una resolución normal de pantalla como 1920x1080 píxeles, se *renderiza* a una bastante menor.

Para la textura de reflexión se posiciona la cámara invirtiendo la altura a la que se encuentra y su ángulo de rotación. Mientras que, para la de refracción, se mantiene la cámara en su posición original como se ve en la figura 3.5.

Figura 3.5: Posicionamiento de la cámara para los dos *renderizados* previos de la escena

Por último, se combinan estas dos texturas aplicando la textura resultante a la superficie del agua durante el *renderizado* final.

Principio de Fresnel: Este principio permite comprender el efecto óptico que se produce al variar el ángulo con el que se observa una superficie reflectante. Si el ángulo de visión se acerca a cero, es decir, a estar paralelo con la superficie, esta es más reflectante. Sin embargo, si el ángulo es perpendicular esta reflejará mucho menos viendo en el caso del agua lo que hay en su interior.

Se realiza el cálculo de este ángulo y dependiendo del resultado se combinarán las texturas de refracción y reflexión de forma distinta. Teniendo un mayor peso la de refracción si el ángulo es muy pequeño y viceversa.

Distorsión: Solo con los efectos anteriores se dispone de un agua totalmente cristalina pero lo que sucede realmente en la mayoría de los casos es que el agua está borrosa debido a su movimiento causado usualmente por el viento. Este efecto se consigue por lo que se conoce como un mapa de distorsión o *DuDv map*.

Un mapa de distorsión es una textura que tiene colores donde solo varían los canales rojo y verde, sin variar el canal azul. Estos dos valores indican la posición que se debe desplazar en la textura original para seleccionar el píxel de color que se va a dibujar en la posición actual. Por ejemplo, si en la posición que nos encontramos hay que dibujar el color de la textura que se encuentra en la posición (0.2, 0.1) y el mapa de distorsión dice de desplazar +0.1 en cada eje. Se va a dibujar finalmente el color que se encuentra en la posición de la textura (0.3, 0.2).

Profundidad: Cuando el nivel del agua aumenta, el color que tiene la superficie refractada se oscurece ya que la luz va atenuándose conforme aumenta la profundidad. Este efecto se descarta en los océanos ya que son demasiado profundos, exceptuando si se encuentran en la orilla o junto a un objeto, por lo que el color resultante sería el negro.

Para implementar esta técnica, lo que se hace es ir oscureciendo la zona observada conforme aumenta la profundidad del agua. Para ello se hace uso del ***Depth Buffer***, este *buffer* permite conocer la distancia a la que se encuentra la cámara de cada elemento en pantalla. Estos datos se almacenan en una textura para cuando se realice el proceso final de *renderizado* de la escena saber a qué profundidad se encuentra cada píxel y oscurecer más o menos los correspondientes píxeles pertenecientes al volumen de agua.

Luz del sol: Para simular los brillos del sol sobre la superficie del agua se utiliza el modelo de iluminación de Phong. Es uno de los más conocidos debido a los buenos resultados que se obtienen con un bajo coste computacional. Este método permite calcular la iluminación de cada píxel o fragmento de pantalla si se conocen las normales de cada vértice. Estas normales se interpolan para obtener un sombreado más suave y menos facetado entre las caras. En concreto, los brillos hacen uso de la iluminación especular que tiene en cuenta como de grande es el ángulo que se genera entre el vector hacia la posición del observador y el vector reflectado del fragmento proveniente de la fuente de luz.

En Bruneton y cols. (2010) se observa otro modelo de iluminación, la función de distribución reflectante bidireccional (del inglés *Bidirectional Reflectance Distribution Function (BRDF)*), que es más realista que el modelo empírico de Phong, ya que tiene en cuenta propiedades físicas como la conservación de la energía. Por este motivo su coste es mayor.

Espuma del mar: Son burbujas que se forman usualmente en las crestas de las olas cuando rompen al encontrarse bastante picadas. Una forma sencilla de implementar esta característica es aplicando una textura con este patrón blanco sobre la superficie del agua y animándolo con desplazamiento y cambio de opacidad. Otra manera es hacer uso de los datos que generan algunos algoritmos para calcular lo picada que se encuentra la ola en

una determinada posición, coloreándola de blanco en caso de que supere cierto umbral; por ejemplo, en Tessendorf (2001) se utiliza la medida Jacobiana que proporciona un valor variable dependiendo del desplazamiento de la ola en ese punto.

3.2.2. Mapeado de normales (*Normal Mapping*)

Hay muchos objetos que tienen superficies rugosas, con pequeñas grietas o deformaciones, y no se pueden añadir vértices para cada desnivel de estos materiales porque se tendrían demasiados y el coste sería demasiado alto. Por ello, simplemente se plasma visualmente en la textura. El problema es que la iluminación de la escena trata toda la textura como una superficie plana reduciendo el realismo al brillar toda la superficie de un material por igual.

Para solventar este problema es para lo que se utiliza el mapeado de normales (de Vries, 2020, Cap.37), empleando una textura para modificar la normal que dispone cada fragmento en una determinada posición de un material. Esto hace que la iluminación se vea afectada por estas pequeñas deformaciones al modificar las normales en estas localizaciones y proporcione una sensación de relieve y mayor realismo. Un ejemplo claro se puede ver en la figura 3.6 donde se utiliza esta técnica reduciendo en gran medida el número de vértices de la malla obteniendo prácticamente un resultado muy similar visualmente.

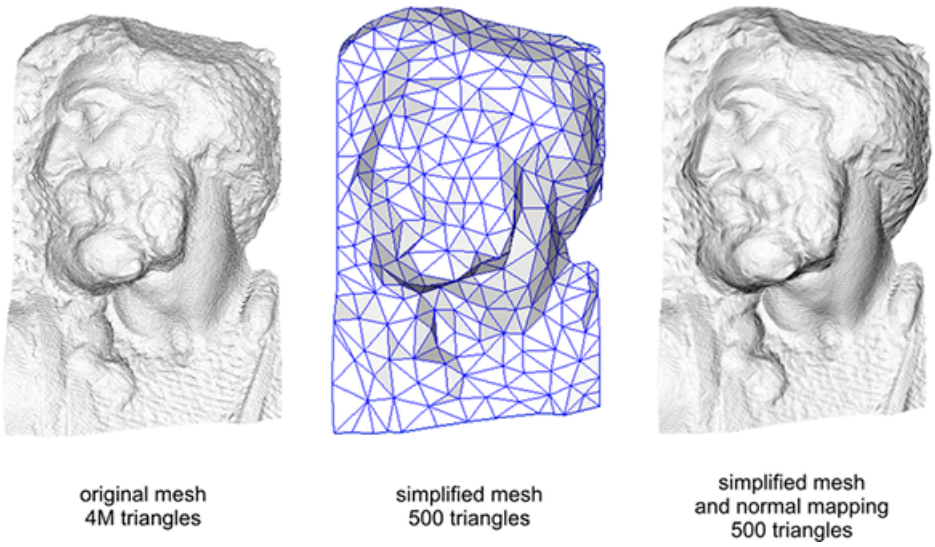


Figura 3.6: *Normal Mapping* aplicado a un modelado para reducir su número de vértices
Fuente: de Vries (2014)

Lo mismo sucede con el océano, está conformado por un gran número de perturbaciones en la superficie del agua, además de las olas tan remarcadas que se generan con los algoritmos vistos en el apartado 3.1. Estas perturbaciones son olas demasiado pequeñas y hay tal cantidad que generarlas con los algoritmos de oleaje modificando la posición de los vértices sería demasiado costoso computacionalmente. Para ello se utilizan los mapas de normales en la textura del agua para simular estas pequeñas variaciones.

Para aplicar los mapas de normales, en lugar de utilizar la normal que viene interpolada

del *Vertex Shader*[1], se escoge la que viene definida en la textura del mapa de normal para cada fragmento. Esta normal se obtiene a través del color en formato *Red, Green and Blue (RGB)*, siendo estos tres canales los que definen el vector X, Y y Z de la normal.

Estos mapas suelen verse de color azul ya que representan la altura con la coordenada Z que corresponde al canal azul, por lo que en la mayoría de los puntos este valor es cercano a uno que es el máximo. Además, se puede dotar a este *normal map* de desplazamiento en la misma dirección que el oleaje para que las pequeñas perturbaciones en el agua parezcan desplazarse junto a este.

3.2.3. *Path Tracing*

El trazado de caminos (del inglés ***Path Tracing***) es un sistema de iluminación de la escena basado en el trazado de rayos que consigue unos grandes resultados realistas. Se puede usar en alternativa al sistema de Phong, pero en videojuegos no se ha comenzado a utilizar hasta hace pocos años debido al gran coste computacional que tiene calcularlo. Explicado en Walt Disney Animation Studios (2016) y en Gentile (2019).

Se comienza lanzando rayos desde la cámara en todas direcciones calculando todos los rebotes con los objetos que inciden hasta que llegan a la fuente de luz. Para evitar que los rayos se queden rebotando demasiado tiempo se suele establecer un número máximo de rebotes por rayo para que en caso de no alcanzar la fuente de luz se descarten. Para mejorar el rendimiento en cada etapa del rebote de los rayos, se agrupan los que llevan una dirección similar para calcularlos juntos, ya que disponen de propiedades similares.

Además, se suelen lanzar unos 12 rayos por píxel, esto va a hacer que la imagen resultante tenga ruido, ya que pueden haber algunos píxeles que no se iluminen. Hay varios métodos empleados para resolver este ruido, por ejemplo, Nvidia utiliza su tecnología *DLSS* basada en inteligencia artificial que trata de eliminar el ruido en cada frame (Harding, 2021). Mientras que Unreal Engine 4, coge varios frames consecutivos para tratar de suavizarlo (Sweeney, 2020).

En el caso del océano solo se busca calcular la iluminación del agua con esta técnica, para ello, en caso de que el rayo de luz colisione contra una superficie opaca se descartará.

Otro método de trazado de rayos es el ***Photon Mapping***, contrario al *Path Tracing*, en el que se envían los fotones desde el elemento que emite luz a los objetos esperando que los rebotes acaben alcanzando la cámara. Dependiendo de la intensidad de la luz se envian más o menos fotones teniendo en cuenta las propiedades de los materiales, y calculando cuantos más se generan a partir de cada rebote.

3.2.4. Efectos cáusticos

Los efectos cáusticos (del inglés *Caustics*) del agua se producen cuando hay poca profundidad y la luz se refracta en el oleaje incidiendo finalmente en el fondo del mar. Al colisionar, los rayos forman zonas con aspecto circular que se ven más claras. Estos efectos no son perceptibles en mar abierto o zonas donde el agua tiene gran profundidad, además suele tener un coste de cómputo bastante elevado. Algunas de las técnicas utilizadas para calcular efectos cáusticos son las siguientes:

[1] Explicado en el apartado 5.2.1.3.

Trazado de rayos realista: Esta técnica descrita en Fernando (2004, Cap.2) se basa en el trazado de rayos permitiendo obtener unos resultados realistas. En lugar de lanzar una gran cantidad de rayos desde la fuente de luz hacia el océano y ver en qué elemento de la escena rebotan se sigue un procedimiento similar al del *Path Tracing* visto en la sección 3.2.3. Se lanzan rayos desde todos los fragmentos del suelo del océano que se encuentran visibles en cámara para que se refracten en el agua y se observe si inciden con la fuente de iluminación de la escena. En caso de que incidan se almacenarán en una textura los fragmentos desde los que se ha partido para aplicarla más adelante sobre la textura del fondo en el proceso de *renderizado*.

Este cálculo se puede realizar desde el *Fragment Shader*[2] obteniendo una mejor calidad a coste de un rendimiento inferior. Además, para evitar un rendimiento variable se *renderiza* con un tamaño de textura fijo pudiendo provocar que no se adapte bien a todos los entornos. Otra mejora es hacer uso del *Depth Buffer* para descartar los rayos de algunos puntos que al estar demasiado profundos no se verán.

Copiado de texturas: Esta propuesta planteada en el siguiente artículo Zucconi (2019), donde se dispone de una textura en escala de grises como la que se aprecia en la figura 3.7. Esta se combina con la textura del fondo oscureciendo las zonas donde la textura se acerca al negro, además de dotarla de movimiento incrementando sus coordenadas de textura para que parezca que se desplaza como el oleaje. Además, se puede combinar con otra textura con un desplazamiento y coordenadas diferentes para obtener un mejor resultado.

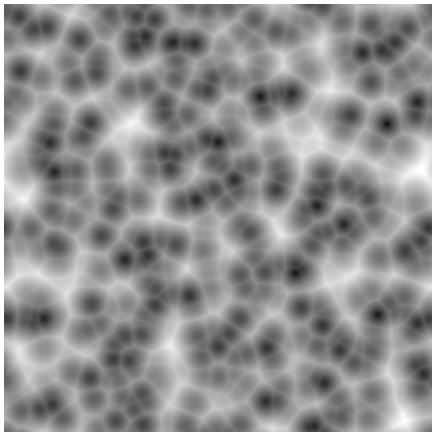


Figura 3.7: Textura cáustica en escala de grises
Fuente: Alisavakis (2019)

Este método trata de simular la iluminación realista pero es totalmente independiente del oleaje aunque el resultado en algunas ocasiones pueda ser el deseado. Es muy eficiente al no tener que realizar ningún cálculo con el oleaje a costa de este realismo.

Doble malla: En Wallace (2016) se utiliza una malla que se sitúa en la luz emisora, que al ser el sol se encuentra posicionada horizontalmente sobre la superficie del agua emitiendo un rayo por cada vértice de la malla. Los rayos se refractan en el agua y se genera otra malla con la misma cantidad de vértices pero con el área de los triángulos que la conforman modificada

[2]Explicado en el apartado 5.2.1.3.

como se puede ver en la figura 3.8. En caso de ser el área de un triángulo mayor o menor en la malla generada que en la original se incrementará o disminuirá el brillo.

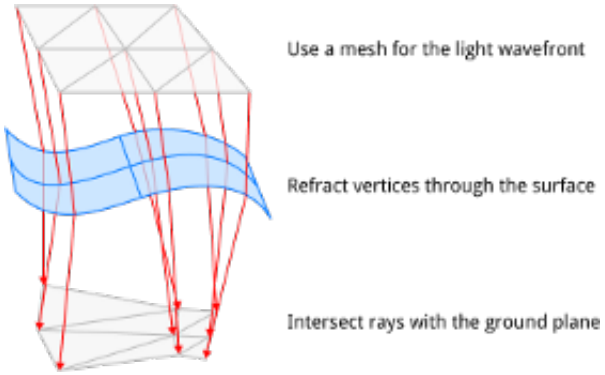


Figura 3.8: Ejemplo de como el rayo se refracta e incide en la segunda malla
Fuente: Wallace (2016)

El problema que tiene es que solo se utiliza con superficies planas, ya que se basa en el cálculo de las áreas de dos dimensiones (2D). Pero se puede evitar rehaciendo el algoritmo para que calcule áreas dentro de un plano en tres dimensiones teniendo que añadir el cálculo de conocer la posición del punto en la malla del suelo donde colisiona el rayo.

El otro problema es que solo se puede realizar con pequeños modelos de agua, ya que el número de rayos que se tiene que enviar para cubrir una gran zona de agua tendría mucho consumo al tener que generar dos mallas adicionales aparte de la del océano. Esto se puede solucionar reduciendo la resolución de la malla dependiendo de la distancia, para que según aumente se reduzca el número de triángulos.

3.2.5. *Renderizado* de sistemas de partículas

Las técnicas de iluminación vistas anteriormente eran aplicadas sobre mallas, para el *renderizado* de partículas es necesario utilizar un método que permita unificar visualmente todas las esferas que componen el sistema. Para ello, en Green (2010) se muestra como se realizan diversos procesos de *renderizado* para obtener finalmente una superficie uniforme con información de su posición y normales en cada punto. Ahora, disponiendo de los mismos datos de los que tiene una malla, se le aplican las técnicas de iluminación deseadas.

3.3. Optimización

Aparte del sistema de mapeado de normales visto en el apartado anterior que se centraba en la reducción de los vértices de la malla utilizando texturas, existen otra gran variedad de métodos que pueden ser utilizados para mejorar el rendimiento de la aplicación. Se van a adaptar para el caso concreto del océano, pero se pueden emplear en otros apartados de un videojuego.

3.3.1. Particionado del espacio

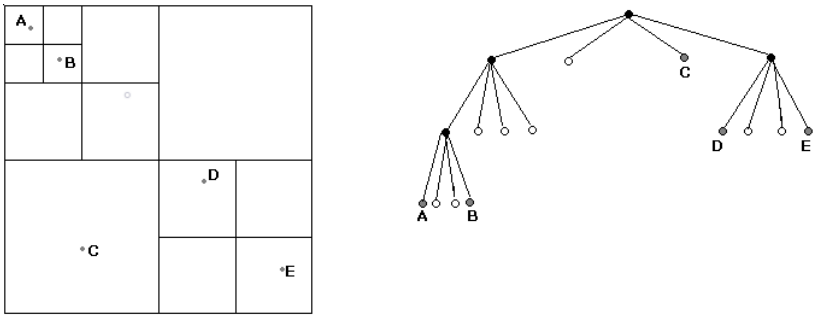
El objetivo principal para el particionado del espacio es evitar que se realicen los cálculos de las animaciones de las olas que no se visualizan en pantalla. Porque, a pesar de que no se vea ningún elemento en pantalla, va a seguir procesándose a no ser que se disponga de alguna forma de controlarlo y decirle cuando debe dejar de calcularse (Rojas, 2013).

Realizar las operaciones para conocer si un elemento se encuentra en pantalla es costoso computacionalmente, si además se emplea para cada uno de todos los elementos que se tiene en nuestro nivel puede llegar a ralentizar en exceso la aplicación. Una forma de solucionar el problema es agrupando estos objetos en conjuntos que se encuentren próximos para luego realizar el cálculo sólo del volumen que los contiene, no para cada objeto.

La forma de solucionar este problema es utilizando los sistemas de particionado del espacio, en los que se va dividiendo el espacio de nuestro nivel en secciones más pequeñas repetidamente conformando una estructura de árbol para disponer de los objetos de la escena agrupados jerárquicamente. Tras esto, se utiliza otra técnica denominada ***Frustum Culling*** con la que es posible calcular cuáles de estas subsecciones se encuentran visibles en pantalla y cuales no se muestran. Al disponer de la estructura de árbol, cuando se descarta una sección se podrá descartar también el resto de las subsecciones contenidas evitándonos realizar los cálculos de estas.

Para aplicar este sistema en el océano se generan mallas de un cierto tamaño establecido en lugar de generar una única malla. Tras esto, se agrupan cada una de las mallas en las secciones en las que se haya dividido el entorno. Calculando con el *Frustum Culling* qué secciones no se encuentran visualizándose y evitando que se realice el cálculo de la animación de las mallas que tienen contenidas.

Por último, hay varios tipos de árboles de particionado del espacio. El *Quadtree* es uno de ellos, se encarga de ir subdividiendo el entorno en cuatro regiones como se puede ver en la figura 3.9. Es la mejor opción para utilizarlo con el modelado del océano, ya que funciona muy bien para videojuegos 2D o videojuegos 3D en los que no se superponen los elementos entre ellos. Debido a que el cálculo de si un cuadrado se encuentra dentro del campo de visión del jugador es más sencillo que si fuese un cubo al disponer de una dimensión menos.



(a) escena dividida con un *quadtree* **(b)** árbol generado de la escena

Figura 3.9: Ejemplo de *Quadtree*
Fuente: Kershaw Winder (2000)

3.3.2. *Level of Detail*

Los Niveles de detalle (del inglés *Level of Detail (LoD)*) se utilizan para mejorar el rendimiento de las aplicaciones disminuyendo el número de vértices de las mallas dependiendo de la distancia a la que se encuentren. De forma que si un elemento se encuentra muy lejos del usuario se sustituye su malla por una con menos resolución siendo imperceptible para el jugador.

El problema se encuentra en mallas como la del océano, que son tan grandes que hay secciones muy próximas al usuario y otras muy alejadas. Por lo que si se busca una buena resolución en una zona pequeña cercana se deben de aumentar los vértices de toda la malla teniendo un impacto drástico en el rendimiento.

Para resolver este problema se utiliza una técnica conocida como ***GeoMipMapping*** que hace uso de un *Quadtree* para reducir el número de vértices de la malla dependiendo de la distancia a la que se encuentre de la cámara, como se puede ver en la figura 3.10. Sin embargo, en las zonas donde cambia esta resolución se suelen generar algunos pequeños problemas visuales, ya que los vértices no se encuentran bien conectados.

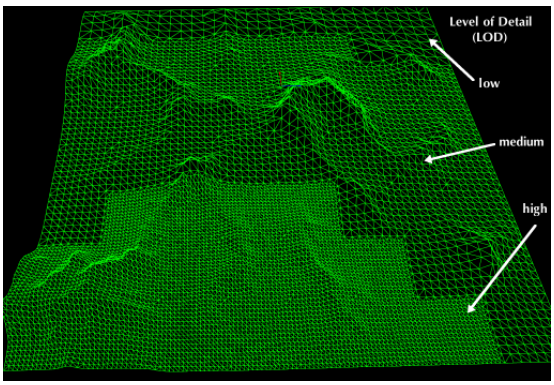


Figura 3.10: *GeoMipMapping* aplicado sobre unas montañas
Fuente: Khan (2013)

En de Boer (2000) se ven algunos de los métodos utilizados para solucionar este problema. Como es el uso de algoritmos para añadir algún vértice extra a la unión de las dos mallas o reordenar la posición de los vértices en la zona donde cambia la malla.

3.3.3. Unidades de procesamiento

Usualmente en los ordenadores domésticos se dispone de dos unidades de procesamiento, la unidad central de procesamiento (del inglés *CPU*) y la unidad gráfica de procesamiento (del inglés *GPU*). Una de las mayores ventajas de la *GPU* es que puede realizar operaciones en paralelo mucho más rápido. Sobre todo es utilizada en videojuegos para el apartado gráfico, ya que una gran cantidad de cálculos para visualizar los elementos por pantalla son totalmente independientes y pueden aprovecharse de esta tecnología para mejorar en gran medida el rendimiento.

Sin embargo, pasar los datos con los que se van a realizar los cálculos de la *CPU* a la *GPU* tiene un determinado coste pero en el caso del apartado gráfico sigue saliendo asequible, ya

que el trabajo computacional ahorrado es mucho mayor al que tiene este procedimiento.

En el caso del proyecto, la *GPU* se puede utilizar para calcular la animación del oleaje siempre que se utilicen algoritmos que se puedan paralelizar. Es decir, que un vértice, por ejemplo, no dependa del resto de vértices de la malla. O que no se requiera el estado anterior, ya que el consumo de pasar mucha información a la *GPU* puede ser mayor de lo que se ahorraría. Por lo que esta técnica se puede utilizar en algunos algoritmos estudiados como el de Gerstner pero es más complicado su uso en sistemas de partículas, ya que los cálculos de estas dependen del resto de volúmenes.

En la tabla 3.1 se puede ver un resumen de los algoritmos analizados.

Algoritmos	**Coste computacional**	**Volúmenes de agua aplicables**	**Dificultad de implementación en *GPU***	**Modelo**
Ondas Compuestas	Bajo	Cualquiera	Bajo	Espacial
Gerstner	Bajo	Cualquiera	Bajo	Espacial
Tessendorf	Medio	Cualquiera	Alto	Espectral
Sistemas de partículas	Alto	Pequeños	Alto	Hidrodinámico
Corrientes de agua en ríos	Medio	Medianos	Alto	Espectral

Tabla 3.1: Características de los algoritmos de oleaje

4. Objetivos

El objetivo general es realizar un análisis de todos los métodos y etapas necesarias para representar gráficamente un océano que pueda utilizarse en un videojuego, así como, el desarrollo de una funcionalidad nueva para el motor gráfico utilizado, implementando alguno de los métodos que se han presentado y realizar un análisis comparativo entre ellos.

A continuación, se definen los objetivos específicos del proyecto:

- Adaptar el motor gráfico desarrollado durante el último año de la titulación para representar océanos, permitiendo realizar su desarrollo en Windows.
- Crear un generador de mallas que sirva como base para los algoritmos de oleaje que se implementen en el proyecto. El generador debe permitir definir el tamaño de la malla y su resolución, es decir, el número de vértices que la conforman.
- Determinar qué algoritmos de oleaje de los estudiados en el estado de la cuestión son más adecuados para incorporar al sistema de representación de océanos y desarrollarlos.
- Seleccionar los efectos de iluminación más adecuados de los estudiados en el estado de la cuestión e incorporarlos al sistema de representación de océanos.
- Determinar los métodos de optimización que mejor se adaptan e implementarlos.
- Diseñar y construir una interfaz que permita configurar los algoritmos, las técnicas de iluminación e incorporar métricas de rendimiento de la aplicación.
- Analizar los resultados de la implementación, comparando el rendimiento de los algoritmos de oleaje, el consumo de las técnicas de iluminación y el ahorro al aplicar los métodos de optimización.

5. Diseño e Implementación

5.1. Diseño del sistema

De los **algoritmos de oleaje** estudiados anteriormente se va a descartar el del flujo del agua en ríos, ya que se va a buscar la representación de un océano y el resultado que proporciona no es tan realista para este, si no para un río o un volumen de agua mediano. Los otros que se descartan van a ser los basados en sistemas de partículas, ya que, como se puede ver en la tabla 3.1 tanto para conseguir un buen rendimiento en una gran superficie de agua como para estudiar en profundidad las ecuaciones empleadas llevaría una gran cantidad de tiempo y su complejidad queda fuera del alcance del proyecto.

El primero de los algoritmos que se va a escoger va a ser el de Tessendorf, ya que los algoritmos basados en las Transformadas de Fourier son los más utilizados en videojuegos. Por ejemplo, el videojuego Sea of Thieves es muy conocido por cómo hace uso del agua utilizando este tipo de algoritmos (Ang y cols., 2018). El segundo va a ser el de ondas Compuestas, ya que como se observa en la tabla 3.1 es uno de los más sencillos de implementar y por donde se puede comenzar a obtener resultados rápidamente debido a tener un mayor conocimiento de sus ecuaciones. Y el tercero, el de Gerstner, va a ser una mejora del anterior, por lo que se va a poder reutilizar casi toda la lógica implementada. Para el algoritmo de Tessendorf se va a poder modificar la dirección del viento en la que se desplaza el oleaje y la altura de este. Para los otros dos, se podrán agregar y eliminar ondas simples modificando sus propiedades de longitud de onda, dirección y amplitud. Además, en el algoritmo de ondas compuestas se va a poder especificar la velocidad de desplazamiento de cada onda simple, y en el de Gerstner, en cambio, se va a poder definir su desplazamiento.

Los **efectos de iluminación** que se van a implementar van a ser la mayoría de los definidos en las propiedades físicas del agua. Como los reflejos y refracciones junto al efecto de Fresnel, pudiendo modificar cuánto se aprecia este efecto en el material. La distorsión del agua, donde se podrá modificar si dispondremos de un agua más agitada o clara. Los brillos del sol definiendo el color de este brillo, y valor con el que el agua comenzará a oscurecerse. También, se implementará el *Normal Mapping* pudiendo establecer el tamaño de las pequeñas olas que se van a generar y la dirección de su desplazamiento. Y, por último, se va a implementar el efecto cáustico del copiado de texturas pudiendo establecer sus dimensiones. Se van a descartar el resto de efectos cáusticos y el trazado de rayos ya que exceden el tiempo planificado para este apartado.

La **técnica de optimización** seleccionada ha sido la de pasar de la *CPU* a la *GPU* los tres algoritmos de oleaje que se van a implementar. Seguramente se hubiese obtenido una mayor mejora de rendimiento con la de particionado del espacio pero ya se ha implementado un método similar en otro proyecto, por lo que se ha priorizado el aprendizaje de nuevas técnicas en este caso.

La **interfaz** va a permitir configurar y visualizar toda la funcionalidad implementada. Va

a ser ajena al motor, pidiéndole los datos necesarios para mostrarlos en pantalla y enviándole los hayan sido modificados por el usuario. Para ello, en primer lugar, se va a disponer de un selector que permita escoger cuál de los algoritmos de oleaje implementados se va a mostrar. Además, de permitir modificar las propiedades de los parámetros modificables definidos en cada algoritmo. También, se van a poder modificar algunas propiedades de la iluminación como la posición del sol, así como, el color de la luz que emite y sus características configurables. Además, se van a poder activar o desactivar las distintas propiedades físicas del agua para poder visualizar cómo afecta el rendimiento cuando no se están procesando. Por último, se mostrarán dos gráficas en las que se mostrará el registro temporal de los *FPS* y el tiempo en milisegundos de cada ciclo del programa.

Para los **análisis**, se va a realizar una comparación de rendimiento de los algoritmos que se van a implementar tanto en la *CPU* como en la *GPU*. Así como, comprobar si el coste de pasar todos los valores a la gráfica es menor que la mejora que se produce al calcularlo en esta. Además de comprobar la escalabilidad que tienen los algoritmos implementados al usarlos en mayores superficies con una mayor cantidad de vértices. Por último, se va a analizar las técnicas de iluminación que tienen un mayor coste.

Los métodos de optimización van implícitos en el motor, por lo que el diseño general resultante se muestra en la figura 5.1 donde se encuentra la interfaz gráfica y el motor gráfico. El motor contiene el módulo de generación de oleaje y el apartado de iluminación. El módulo de generación de oleaje se encuentra conformado por el generadorde mallas y los algoritmos de oleaje.

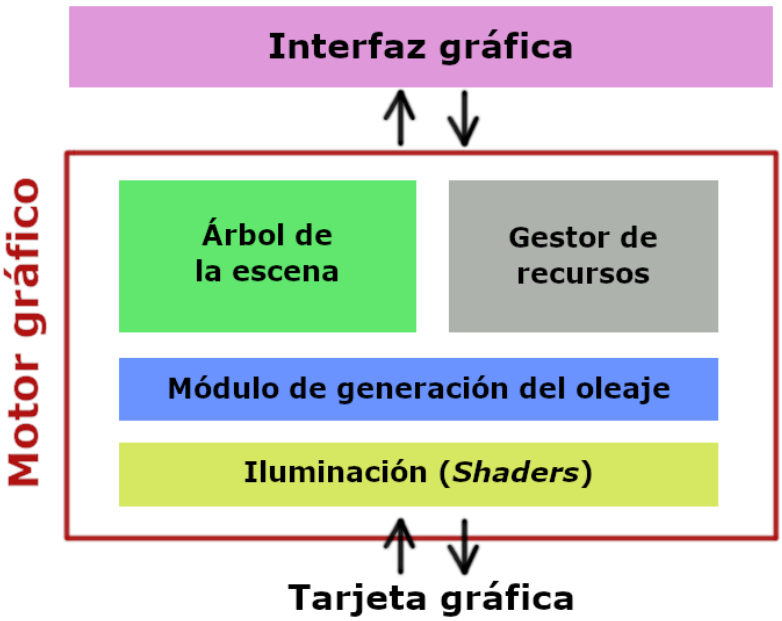


Figura 5.1: Diseño de la aplicación

5.2. Motor gráfico

Un motor gráfico es un *software* de alto nivel que abstrae al usuario que lo utiliza de las comunicaciones a bajo nivel que realiza con la gráfica para dibujar elementos en pantalla. Se encuentra conformado por una serie de sistemas que se encargan del manejo y la estructuración de los objetos en la escena, además de incorporar al apartado gráfico técnicas de coloreado, iluminación, sombreado, animación, etc.

Los ***Shaders*** son programas que se ejecutan en la gráfica, donde se realiza el cálculo de estas técnicas gráficas para cada vértice junto con los datos provenientes de la aplicación, como pueden ser las propiedades de las luces o las posiciones de los elementos en la escena. Los *shaders* se emplean debido a que se puede aislar el cálculo de cada vértice de una malla, pudiendo paralelizarse para mejorar el rendimiento como se vio en la sección 3.3.3. Se programan con un lenguaje de programación de shaders como es *OpenGL Shading Language (GLSL)*, que es el estándar utilizado por OpenGL, además se compilan al inicio de la aplicación. En la sección 5.2.1.3 se explica extendidamente acerca de los distintos tipos de *shaders* que hay y cuál es el proceso que recorre la información hasta que se muestra en pantalla.

5.2.1. Arquitectura principal

A continuación, se describen las características principales que conforman el motor.

5.2.1.1. Árbol de la escena

El árbol de la escena contiene la información necesaria para posicionar cada elemento en esta, para ello almacena los datos de la posición, rotación y escalado de cada uno. **CLNode** es la clase que guarda esta información necesaria para posicionar cada objeto que se encuentra ordenado en una estructura de árbol jerárquica que permite tener clases heredadas dependientes de las transformaciones del padre.

CLNode, además, va a contener la clase **CLEntity** que va a ser la clase padre de la que hereden los distintos elementos que haya en la escena como pueden ser las mallas, las luces o las cámaras entre otros. Esta herencia es de utilidad a la hora de realizar agrupaciones y poder llamar a métodos comunes entre los hijos de CLEntity. Otra información almacenada por CLNode es la de los *shaders* que se aplican a cada elemento, pudiendo así aplicar distintos por cada uno si es necesario. En la figura 5.2 se puede ver como se estructura la información en la clase CLNode.

Para realizar los cálculos de las posiciones de los objetos se trabaja con matrices de tamaño 4x4 homogéneas que aportan una gran facilidad a la hora de combinar transformaciones. La matriz de transformación, almacenada en la clase CLNode, es el resultado de combinar las matrices respectivas a la translación, rotación y escalado. Aplicando esta matriz al objeto se puede determinar su transformación en la escena, y en caso de ser una clase heredada se multiplicará la matriz por la del elemento superior en la jerarquía y así sucesivamente. En el anexo A se puede repasar como se genera esta matriz resultante.

Actualmente, se dispone de la transformación del objeto respecto a un punto que se ha definido como el inicial, esto genera dificultades a la hora de dibujar los objetos en pantalla al no situarse la cámara en el eje de referencia. Para facilitar el proceso se van a posicionar los objetos respecto a la cámara siendo esta el centro de la escena. Para ello se calculan las

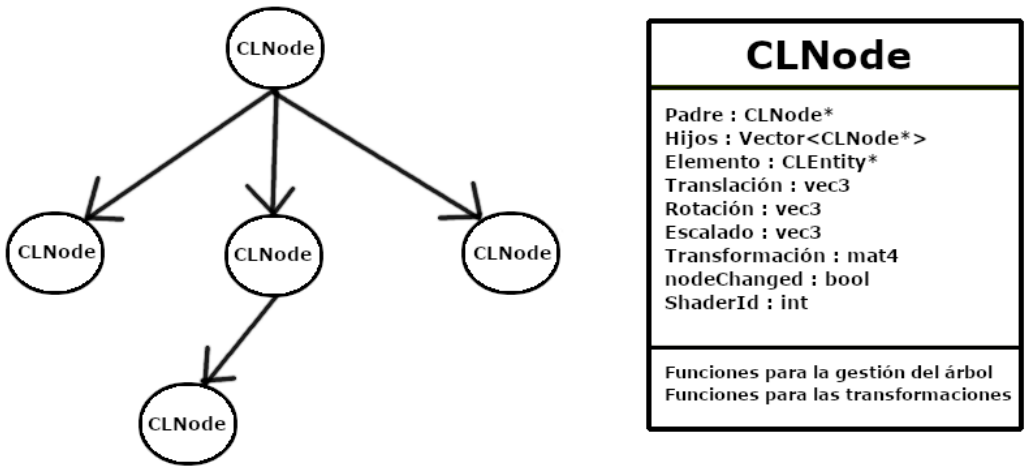


Figura 5.2: Estructura de la clase CLNode

matrices de transformación necesarias para pasar, en primer lugar, de coordenadas de mundo a coordenadas de cámara y de estas a coordenadas de proyección, ya que la perspectiva con la que se visualizan los objetos puede ser distinta. Estas dos matrices que se van a calcular van a ser las matrices de vista (del inglés ***view***) y de proyección (del inglés ***projection***), se puede ver una ampliación de como obtenerlas en el anexo B.

5.2.1.2. Gestor de recursos

El gestor de recursos es el sistema que permite gestionar la carga de elementos externos al código como pueden ser imágenes, modelados e incluso *shaders*. Realiza un control de la memoria evitando que se vuelva a cargar contenido que ya se ha cargado con anterioridad o descargando elementos que se han dejado de utilizar.

Se ha implementado junto a la librería Assimp que nos permite extraer la información de los vértices, normales y texturas de los modelados utilizados en el proceso de *renderizado*. Cuando se añade un nuevo modelado se crea la clase **CLMesh** que es una herencia de CLEntity añadida al árbol de la escena. CLMesh tiene una instancia de la clase **CLResourceMesh** donde se almacena la información de la malla y otra de **CLResourceMaterial** donde se guarda el material. También, se dispone de una estructura como **CLResourceTexture** y **CLResourceShader** para almacenar la información relativa a las imágenes y a los *shaders*.

5.2.1.3. *Pipeline* gráfico

Un ciclo de juego se divide en tres etapas o fases para cada elemento de la escena que se desee dibujar:

- La fase del cálculo de la **lógica del juego** donde se realiza el cálculo de las físicas, la lógica o inteligencia artificial de este elemento. Esta etapa es independiente del motor gráfico.

- En la fase del ***draw*** se llaman a los comandos de OpenGL que se encargan de pasar la información necesaria a la *GPU* para realizar el cálculo de los *shaders* como pueden ser los vértices de las mallas. En esta etapa, también, se realiza el cálculo de las matrices de transformación, vista y proyección, necesarias para el objeto actual. Además, de agregar las matrices de otros elementos como las luces o texturas que influyen en el proceso de *renderizado*.

- La fase del ***render*** es la fase donde se ejecutan los *shaders* con los datos pasados en la etapa anterior con un orden específico, esto se conoce como *Pipeline gráfico*.

El ***Pipeline* gráfico** es el proceso ordenado por el que pasan los vértices enviados desde la etapa del *draw* hasta que se dibujan en pantalla. Estos vértices pasan por una serie de etapas antes de visualizarse, estas etapas se pueden ver en la figura 5.3. El *vertex*, *fragment* y *geometry shader* son los que se han implementado para visualizar cada elemento, en el resto de las etapas es OpenGL el que se encarga de procesar la información. A continuación, se va a describir qué funcionalidad se desarrolla en cada etapa:

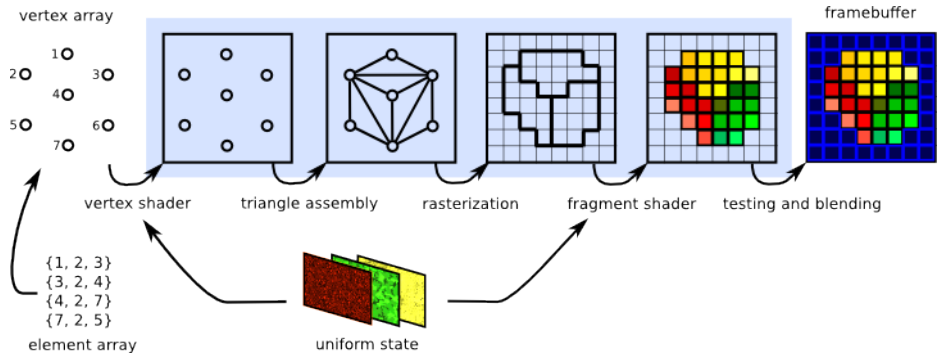


Figura 5.3: Etapas del *pipeline* gráfico
Fuente: Dunn y Wood (2016, Cap.2)

- ***Vertex Shader*:** Se ejecuta por cada vértice, coge la posición del vértice en el modelo y se le aplican las matrices de transformación, vista o proyección para obtener el vértice en el sistema de coordenadas que se desee. Usualmente se desea obtener en coordenadas de proyección, para ello se pasan las tres matrices ya multiplicadas para solo tener que realizar la multiplicación por la posición del vértice.

- ***Triangle assembly*:** Ensambla todos los vértices provenientes de la etapa anterior.

- ***Geometry Shader*:** Se encarga de añadir vértices adicionales al elemento. No aparece en la imagen del pipeline porque se utiliza muy poco debido a su mal rendimiento. En este proyecto solo se ha utilizado para tareas de depuración.

- **Rasterización:** Convierte el modelo abstracto matemático del objeto en los píxeles que lo componen en pantalla, asignando los píxeles que le corresponden a cada triangulo del objeto. A la asignación de cada píxel a cada triángulo de la escena se le denomina fragmento.

- ***Fragment*:** Por cada fragmento que se ha obtenido de la fase anterior le asignamos un color. Es donde se realizan todos los cálculos de iluminación y sombreado para obtener el color resultante. Teniendo en cuenta las luces de la escena, texturas, normales, efectos, etc.

- ***Blending*:** Etapa final que combina todos los elementos que se han enviado a dibujar en pantalla y muestra el resultado en esta. Se tiene en cuenta la profundidad a la que se encuentra el fragmento respecto al resto de objetos para dibujarlo o descartarlo en caso de que se esté ocluyendo por otro elemento.

En el código 5.1 podemos ver una representación de cómo sería el ciclo del juego.

Código 5.1: Pseudocódigo en C++ para representar el bucle del juego

```
// game loop
int main(){
    while(pulsarEscape){
        calcularLogicaJuego();
        actualizarInterfaz();
        recorrerArbol(entidadPrincipal);
    }
    return 0;
}

// recorremos el arbol y para cada elemento
void recorrerArbol(CLEntity* entidad){
    entidad->calcularYpasarDatosGPU();
    entidad->comenzarADibujar();
    for(CLEntity* hijo : entidad->GetHijos())
        recorrerArbol(hijo);
}
```

5.2.2. Entorno de desarrollo

Como se ha comentado al comienzo del documento, para el desarrollo del proyecto se ha utilizado el motor gráfico SparkEngine[1] desarrollado junto a mi grupo de trabajo en el último año de la titulación. El motor y sus librerías se compilaban directamente desde Windows haciendo uso de *makefiles*[2], se decidió llevarlo a MVS para facilitar la compilación y agilizar el proceso de desarrollo. Durante la fase inicial del proyecto se descargaron y agregaron a MVS una serie de librerías necesarias para el funcionamiento del motor gráfico:

- **OpenGL**: Interfaz de programación de aplicaciones (API) multiplataforma que permite la comunicación con el hardware del dispositivo, usualmente la tarjeta gráfica, para poder visualizar gráficos 2D o 3D. La aplicación se desarrolla con la versión 4.5 de OpenGL.

- **GLEW**: Librería multiplataforma que se encarga de comprobar en tiempo de ejecución que extensiones de OpenGL son soportadas en la plataforma objetivo.

[1] Enlace al repositorio de GitHub de **SparkEngine**: `https://github.com/JoseM98/Spark-Engine`

[2] Los ***makefiles*** son archivos que contienen instrucciones que dicen como se debe de compilar cada uno de los archivos de código.

- **GLFW**: Librería para el manejo de ventanas donde se van a mostrar los gráficos procesados con OpenGL, además de la detección del input del usuario.
- **GLM**: Librería matemática que incorpora clases y funciones que facilitan el manejo de OpenGL incorporando características como vectores, matrices y otras operaciones típicas entre estas.
- **Assimp**: Librería que se utiliza para la importación de modelados 3D, transformado la información proveniente de diferentes formatos de archivos de modelado 3D, en datos con un formato estándar con los que se puedan trabajar. Se utiliza para la lectura de los modelados que se vayan a usar.
- **stb_image**: Librería utilizada para la carga de imágenes en C++. Utilizada junto a OpenGL para cargar las imágenes de los modelados 3D.
- **Dear ImGUI**: Librería gráfica que incorpora distintas herramientas para la gestión de la interfaz del usuario como pueden ser *sliders*, botones o casillas para introducir texto. Estas funcionalidades permiten agilizar la depuración y la interacción en tiempo real con la aplicación.
- **ImPlot**: Librería que extiende de ImGUI para poder representar visualmente datos con el uso de una gran variedad de gráficas.

Una vez que se encontraba el motor completamente funcional se ha creado un mapa vacío con este dónde poder visualizar por pantalla los elementos creados, realizar pruebas y comenzar a desarrollar la aplicación.

5.3. Módulo de generación de oleaje

La funcionalidad de oleaje que implementemos en el motor debe ser lo suficientemente versátil para disponer de distintas técnicas de iluminación y oleaje siendo fácilmente escalable. Por ello, el diseño planteado como se puede ver en la figura 5.4 es el de una herencia; Siendo la clase principal, **CLWaveSystem**, la que incorpora las funcionalidades comunes a los algoritmos de oleaje, como los datos de la malla generada o de los identificadores de los *shaders* que se van a aplicar. Se busca que las técnicas de iluminación desarrolladas en los *fragment shaders* puedan ser aplicadas a cualquiera de los algoritmos desarrollados.

Las clases heredadas de CLWaveSystem van a ser los algoritmos de oleaje que implementen las estructuras de datos y métodos necesarios para realizar el cálculo y modificar los datos de la malla generada. Además, se ha agregado un método *template* que facilita a los usuarios crear instancias de los distintos algoritmos de oleaje simplemente definiendo las propiedades comunes a la clase principal; Generándose con unos valores por defecto modificables para los datos particulares de cada sistema.

5.3.1. Generador de la malla

Los valores necesarios para generar la malla pasados como argumentos al crear el sistema de oleaje van a ser el tamaño y la densidad de la malla. De forma que se definan las dimensiones deseadas y cuanta resolución (número de vértices) tendrá la propia malla.

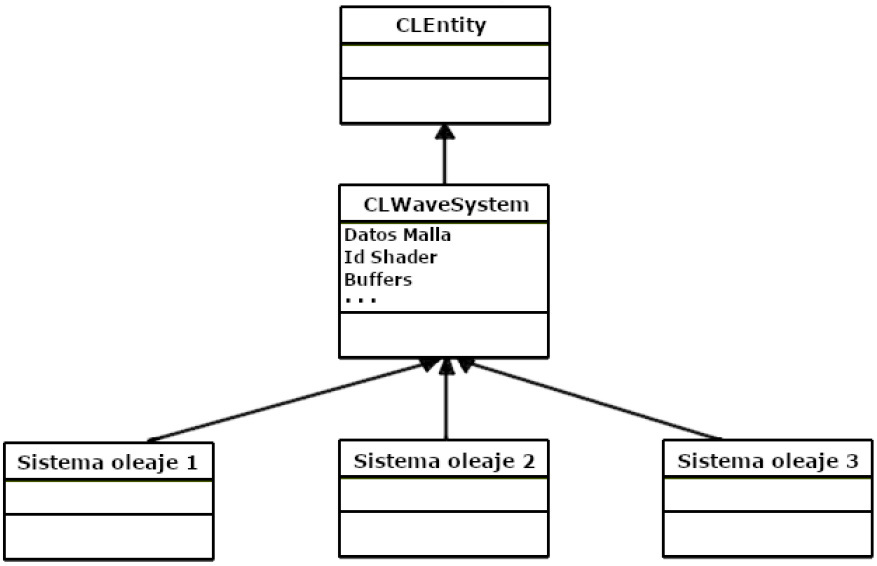


Figura 5.4: Estructura de clases implementada, donde CLWaveSystem hereda de la clase CLEntity y los algoritmos de oleaje heredan de CLWaveSystem

El sistema de generación almacena en una estructura de datos la información de dónde se posicionan los vértices de la malla y las normales que tiene cada uno, inicialmente apuntando hacia arriba. Y en otra estructura los índices de los vértices que conforman cada triángulo. Tras esto, se van a colocar estos datos en los *buffers* necesarios para enviar la información al proceso de *renderizado*, pudiéndose actualizar en caso de calcular los algoritmos y modificar los datos de los vértices. Se hace un almacenamiento previo, ya que cuando se desarrollen los algoritmos en la *CPU* no se va a realizar ninguna actualización de los vértices antes de enviarlos a la gráfica.

5.3.2. Algoritmos de oleaje

La complejidad de este apartado se encuentra en los conocimientos matemáticos y físicos necesarios para obtener las ecuaciones resultantes aplicando los datos correctamente en estas, más que en el desarrollo de la estructuración o arquitectura del código.

Cada uno de los siguientes algoritmos se han creado como una clase derivada de CLWaveSystem. Cuando se recorre el árbol de la escena enviando a dibujar todos los elementos se realiza el cálculo necesario para cada uno especificado en los métodos de la clase y se envían los datos a dibujar a la gráfica. Se ha decidido realizar los cálculos en la fase de *draw* en lugar de en la de cálculo de la lógica del juego, ya que se tiene previsto mover todos los cálculos a la gráfica evitando temporalmente tener que recorrer el árbol de la escena dos veces.

5.3.2.1. Ondas compuestas

Para el algoritmo de ondas compuestas se parte de la ecuación de la onda simple que se desplaza en una dirección, modificándola para que sea posible realizar el cálculo en dos dimensiones. Para ello, la variable que representaba la posición del eje en el que se desplaza

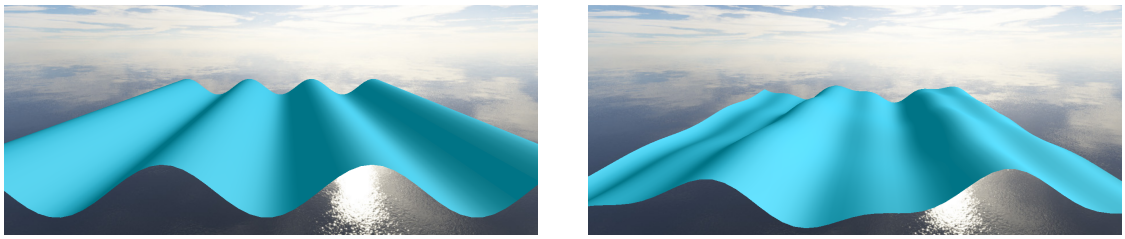
se va a calcular ahora realizando un producto escalar entre la dirección de la ola y su posición en los ejes X y Z, obteniendo la ecuación 5.1 como resultante.

$$P_y = A\sin(\frac{2\pi}{\lambda}(D \cdot (x,z) + vt)) \tag{5.1}$$

Para calcular las normales en los vértices va a ser necesario calcular la derivada respecto a la ecuación analítica anterior en cada uno de los ejes X y Z para obtener sus tangentes. Se va a realizar el producto vectorial entre estas tangentes para obtener las siguientes ecuaciones 5.2 con las que calcular la normal en un punto. En Fernando (2004, Cap.1) se detalla el proceso de obtención de estas ecuaciones.

$$\begin{aligned} N_x &= -\frac{2\pi}{\lambda}AD_x\cos(\frac{2\pi}{\lambda}(D \cdot (x,z) + vt)) \\ N_y &= 1 \\ N_z &= -\frac{2\pi}{\lambda}AD_z\cos(\frac{2\pi}{\lambda}(D \cdot (x,z) + vt)) \end{aligned} \tag{5.2}$$

Aplicando estas ecuaciones de una onda simple a una malla se obtiene el resultado que puede verse en la figura 5.5a, dibujándose con un *shader* básico. Ahora bien, para un resultado menos uniforme se va a realizar el cálculo con varias ondas aplicando un sumatorio a estas ecuaciones y dividiendo por el número de ondas que se dispone. Con unos valores adecuados a cada onda se puede obtener el resultado que se ve en la figura 5.5b. El usuario es capaz de agregar o eliminar ondas simples para poder obtener la onda compuesta deseada.



(a) Resultado aplicando solo una onda simple **(b)** Resultado combinando varias ondas simples

Figura 5.5: Oleaje generado con el algoritmo de ondas compuestas

Tras esto, se han pasado las ecuaciones a un *vertex shader*, se puede ver el código *GLSL* en el anexo C. El problema que ha surgido al hacer este cambio es que OpenGL no admite o es muy complejo el pasar elementos con tamaño variable a los *shaders*. Por lo que en lugar de tener una cantidad indeterminada de ondas se va a tener un *array* con una longitud máxima y una variable que controle cuantas de estas ondas hemos agregado. Enviando así un número fijo a los *shaders* y evitando iterar con más ondas de las que se dispone. La tabla 5.1 muestra las variables definidas en la clase **CLWS_SinWave** necesarias para aplicarse a estas ecuaciones.

5.3.2.2. Gerstner

Con Gerstner se busca obtener un desplazamiento en los ejes X y Z, aparte de en el eje Y, esto se consigue aplicando la onda Trochoidal. Para ello, hay que realizar pequeñas

Variable	En la función	Tipo de dato	Descripción
waveLength	λ	float[maxNumWaves]	Longitud de onda
waveSpeed	v	float[maxNumWaves]	Velocidad de la ola
waveAmplitude	A	float[maxNumWaves]	Amplitud de la ola
waveDirection	D	vec2[maxNumWaves]	Dirección de la ola
timeElapsed	t	float	Tiempo transcurrido, heredado de CLWaveSystem
numWaves		int	Numero de ondas actual
maxNumWaves		const int	Máximo numero de ondas

Tabla 5.1: Tabla con las variables definidas en la clase CLWS_SinWave

modificaciones en la ecuación analítica 5.1, incluyendo una suma del desplazamiento que sigue la circunferencia en la posición actual y la dirección D con la que se modula la dirección de desplazamiento en un determinado eje. Además, se incorpora el factor de desplazamiento Q para permitirnos modular como de agudas son las crestas de las olas. Por lo que las ecuaciones resultantes son las siguientes 5.3.

$$\begin{aligned} P_x &= x + QAD_x \cos(\frac{2\pi}{\lambda}(D \cdot (x,z) + vt)) \\ P_y &= A \sin(\frac{2\pi}{\lambda}(D \cdot (x,z) + vt)) \\ P_z &= z + QAD_z \cos(\frac{2\pi}{\lambda}(D \cdot (x,z) + vt)) \end{aligned} \tag{5.3}$$

Ahora, para calcular las ecuaciones de la normal en ese punto se realiza el mismo proceso que en el algoritmo anterior, calculando las tangentes aplicando las derivadas de las ecuaciones obtenidas respecto a los ejes X y Z, para posteriormente aplicar el producto vectorial. Con esto se obtienen las siguientes ecuaciones que calculan la normal en un punto 5.4. Por último, para hacer dependiente la velocidad el oleaje de la gravedad se ha sustituido la variable de la velocidad por el cálculo $v = \sqrt{\frac{g\lambda}{2\pi}}$. Una explicación más detallada de la obtención de las fórmulas se puede ver en Fernando (2004, Cap.1).

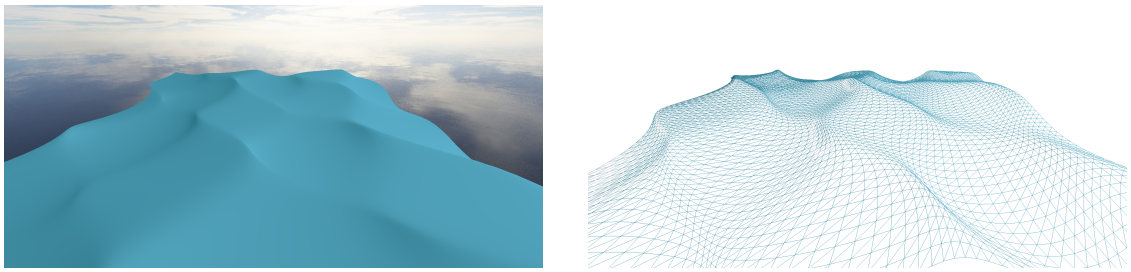
$$\begin{aligned} N_x &= -\frac{2\pi}{\lambda} AD_x \cos(\frac{2\pi}{\lambda}(D \cdot (x,z) + vt)) \\ N_y &= 1 - \frac{2\pi}{\lambda} AQ \sin(\frac{2\pi}{\lambda}(D \cdot (x,z) + vt)) \\ N_z &= -\frac{2\pi}{\lambda} AD_z \cos(\frac{2\pi}{\lambda}(D \cdot (x,z) + vt)) \end{aligned} \tag{5.4}$$

En cuanto a la estructura, se ha creado la clase **CLWS_GerstWave** derivada de CLWaveSystem. La estructura es muy similar al método anterior, ya que se han utilizado los mismos métodos aplicando las modificaciones descritas, se ha eliminado la variable de la velocidad y se han agregado las variables necesarias descritas en la tabla 5.2. Tras esto, se ha repetido el proceso y se ha llevado el cálculo del procesador a la gráfica, en el anexo D se muestran las funciones modificadas en el nuevo *shader*. El resultado de aplicar este *shader* se puede ver en

la figura 5.6.

Variable	En la función	Tipo de dato	Descripción
gravedad	g	float	Gravedad de la Tierra
Qdisplacement	Q	float[maxNumWaves]	Factor de incremento del desplazamiento lateral

Tabla 5.2: Tabla con las variables nuevas añadidas a la clase CLWS_GerstWave junto a las que ya se tenían, mostradas en la tabla 5.1



(a) Empleando un *shader* básico **(b)** Mostrando los vértices de la malla

Figura 5.6: Resultado aplicando el algoritmo de Gerstner

5.3.2.3. Tessendorf

Para obtener la ecuación de Tessendorf que permite calcular el mapa de alturas se parte de la ecuación de Euler $e^{ix} = \cos x + i \sin x$ aplicándose a la ecuación de la onda, obteniendo la fórmula 5.5 con la que obtener la altura en un punto. Para realizar cálculos en el dominio de la frecuencia es necesario el uso de números complejos, por lo que el cálculo de $\tilde{h}(\vec{k}, t)$ permite obtener un número real mediante la siguiente ecuación 5.6, donde $\tilde{h}_0(\vec{k})$ es la función que permite obtener la altura compleja utilizando los valores gaussianos y la función de Phillips que los regula. Y $\omega(k)$ es la función de dispersión que relaciona la longitud de onda y la gravedad.

$$h(\vec{x}, t) = \sum_{\vec{k}} \tilde{h}(\vec{k}, t) \exp(i\vec{k} \cdot \vec{x}) \tag{5.5}$$

$$\tilde{h}(\vec{k}, t) = \tilde{h}_0(\vec{k}) \exp(i\omega(k)t) + \tilde{h}_0^*(-\vec{k}) \exp(-i\omega(k)t) \tag{5.6}$$

Con el cálculo de la derivada de 5.5, se obtiene la ecuación para calcular la normal del vértice en un punto es 5.7. De un modo similar a como se hace con Gerstner se va a calcular

un desplazamiento para los ejes X y Z empleando la fórmula 5.8.

$$\begin{aligned} N_x &= -\sum_{\vec{k_x}} i\vec{k_x}\tilde{h}(\vec{k},t)\exp(i\vec{k_x}\cdot\vec{x}) \\ N_y &= 1 \\ N_z &= -\sum_{\vec{k_z}} i\vec{k_z}\tilde{h}(\vec{k},t)\exp(i\vec{k_z}\cdot\vec{x}) \end{aligned} \tag{5.7}$$

$$\vec{D}(\vec{x},t) = \sum_{\vec{k}} -i\frac{\vec{k}}{k}\tilde{h}(\vec{k},t)\exp(i\vec{k}\cdot\vec{x}) \tag{5.8}$$

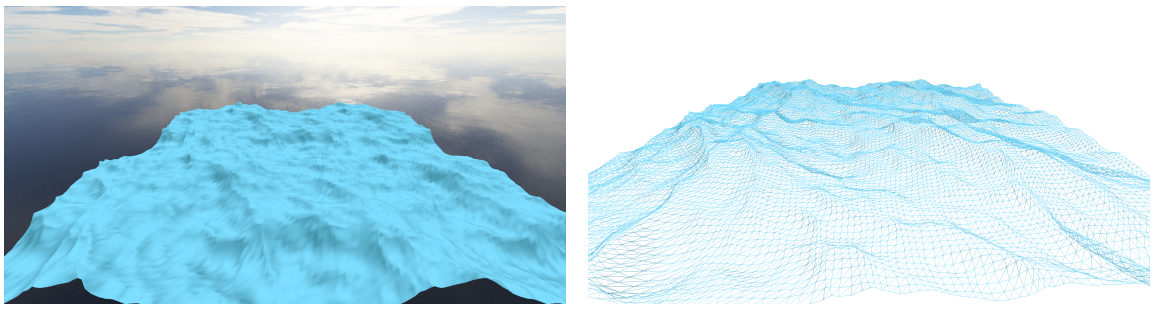
En una primera instancia, se ha realizado el cálculo con la Transformada Discreta de Fourier iterando en cada vértice por todos los de la malla con un coste n^2, tras comprobar que todo funcionaba correctamente se ha pasado a calcular empleando la Transformada Rápida de Fourier iterando solo sobre una parte de ellos reduciendo el coste a $n \log n$. Tras esto se ha tratado de pasar el procedimiento de la *CPU* a la *GPU* pero debido a su complejidad queda fuera del alcance del proyecto.

En cuanto a la estructuración, se ha generado la clase **CLWS_Tes** heredada de CLWaveSystem definiendo una serie de variables, las que nos permiten modificar el aspecto visual de la ecuación se encuentran especificadas en la tabla 5.3, con las que realizar todos los cálculos necesarios. Pudiendo modificar lo picadas que se encuentran las olas o dirección de desplazamiento. La función de Phillips solo es calculada al comienzo, por lo que si se modifican algunos de los valores definidos en la tabla es necesario recalcularla.

Variable	Tipo de dato	Descripción
gravedad	float	Gravedad de la Tierra utilizada para calcular la dispersión $\omega(k)$
philParam	float	Parámetro que afecta al cálculo del espectro de Phillips y hace que varie lo agitadas que se encuentran las olas
waveDirection	vec2	Dirección del viento que se utiliza para calcular la velocidad V en la fórmula
waveLength	float	Longitud de onda, se utiliza para el cálculo de k

Tabla 5.3: Tabla con las variables definidas en la clase CLWS_Tes

Para mejorar el rendimiento se puede calcular el mapa de alturas para una pequeña superficie y esta duplicarla tantas veces como sea necesario. Para evitar que se produzcan cambios bruscos en las uniones la longitud de onda debe ser un múltiplo del tamaño de la malla. En la figura 5.7 se puede apreciar el resultado para cuatro secciones iguales. Estableciendo un gran número de secciones duplicadas puede llegar a ocasionar un resultado muy repetitivo si se observan gran cantidad de estas simultáneamente.



(a) Empleando un *shader* básico **(b)** Mostrando los vértices de la malla

Figura 5.7: Resultado aplicando el algoritmo de Tessendorf

5.4. Iluminación

A continuación, se van a especificar todas las técnicas de iluminación desarrolladas. Estas técnicas se van a implementar en un *fragment shader* común al que se le van a pasar los datos de los vértices procesados en los *vertex shader* de cada técnica de oleaje. Para ello, la información enviada desde cada *vertex shader* dispone de una estructura concreta.

Se parte del modelo de iluminación de Phong, en el que se calcula cómo se refleja la luz en cada fragmento de la escena utilizando la siguiente fórmula 5.9 donde solo se tiene en cuenta la iluminación directa y no las reflexiones de segundo orden. Los datos que se envían desde la *CPU* para realizar el cálculo son: la posición del observador, la posición de la fuente de luz, las propiedades de la luz como la intensidad o atenuación, y las propiedades del material del objeto.

$$\begin{aligned} I_{Phong} &= I_{ambiental} + I_{difusa} + I_{especular} \\ I_{Phong} &= I_a k_a(\lambda) + I_d k_d(\lambda)(L \cdot N) + I_e k_e(\lambda)(R \cdot V)^n \end{aligned} \tag{5.9}$$

5.4.1. *Skybox*

Se conoce como *SkyBox* a la técnica usada para representar el cielo en el videojuego empleando un cubo de seis caras. Esta técnica es fundamental para implementar el reflejo en el agua, ya que el cielo va a ser la mayor parte de lo que se vea reflejado en el océano. El motor ya implementaba esta técnica pero haciendo uso de unas texturas demasiado *cartoon*, por lo que se han cambiado las seis texturas por las que se observan en la figura 5.8 con un estilo más realista. En de Vries (2020, Cap.27.2) se detalla como implementarlo.

5.4.2. Reflejos y refracciones

Se comienza obteniendo las texturas del reflejo y la refracción, para ello se *renderiza* la escena en dos ocasiones almacenando cada resultado en una textura. En cada *renderizado* la cámara se situa en una posición diferente para simular el reflejo y la refracción respecto a la posición original de la cámara. Una vez almacenadas las texturas, se envían al *fragment shader* para que se apliquen en el proceso de *renderizado* de la malla del océano, en el código 5.2 se muestra como se estructura el orden de las iteraciones de *renderizado* enviando los parámetros necesarios.

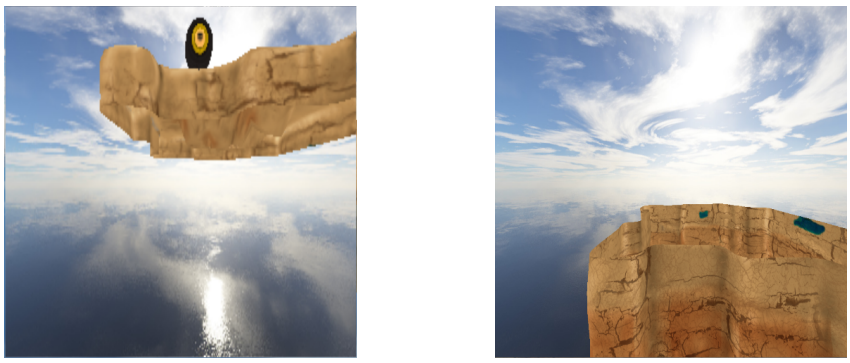
Figura 5.8: *Skybox* utilizaddo en el proyecto
Fuente: de Vries (2020)

Código 5.2: Estructura de los distintos *renderizados* que se hacen

```
glm::mat4 VPmatrix;
CLNode* camera = GetActiveCameraNode();

MoveReflectionCameraPosition(camera);
DrawReflectionTexture(glm::mat4(1.0f), GetActiveCamera(), VPmatrix, actualWaveSystN);

MoveRefractionCameraPosition(camera);
DrawRefractionTexture(glm::mat4(1.0f), GetActiveCamera(), VPmatrix, actualWaveSystN);

DrawSkybox();
// se dibujan todos los elementos del árbol
smgr->DFSTree(glm::mat4(1.0f), GetActiveCamera(), VPmatrix);
```

Al modificar las posiciones de la cámara es posible que algún elemento se sitúe por delante de esta y no permita visualizar la superficie del agua correctamente generando un problema al almacenar una imagen incorrecta en estas texturas. Por ello, se aplica en el *vertex shader* la técnica de ***clipping*** sobre la superficie del agua, permitiendo descartar todos los vértices que se sitúan a un lateral de un plano definido. Para la textura de reflexión se aplicará el *clipping* al plano del agua descartando todos los vértices de la parte inferior de este, y para la de refracción se descartan todos los vértices de la parte superior del plano permitiendo, además, mejorar el rendimiento al procesar menos vértices. En la figura 5.9 se observan las texturas almacenadas.

Tras esto, se envían las texturas obtenidas al *fragment shader* que *renderiza* la malla del océano. Surge un problema al tratar de relacionar las coordenadas 2D de las texturas generadas con las posiciones de la malla del océano para poder dibujar las texturas sobre esta. Para poder realcionarlas se utiliza la técnica de ***Projective Texture Mapping*** (Everitt, 2001) con la que se proyectan estas texturas sobre la escena para conocer el punto correcto en el que dibujar cada una. Este método se puede ejemplificar como si se tuviese un proyector



(a) Textura reflejada a una resolución de 320x180 píxeles

(b) Textura refractada a una resolución de 1280x720 píxeles

Figura 5.9: Texturas reflejada y refractada obtenidas con distintas resoluciones
Modelado 3D de la roca obtenido en Ohorodnichuk (2018)

situado en la posición de la cámara y al proyectarse el dibujo sobre la escena este se deformase como se puede ver en la figura 5.10. Esta técnica va a convertir la posición del fragmento en la escena que se encuentran en espacio de proyección a un espacio normalizado mediante la división de sus coordenadas por la cuarta componente, proporcionando unas coordenadas en los ejes X y Z que se encuentran acotadas en el espacio (-1, 1). Tras esto, se ajusta el espacio obtenido de (-1, 1) a (0, 1) para que sea el mismo al utilizado por las texturas enviadas al *shader*. Este proceso se puede ver en el código 5.3.

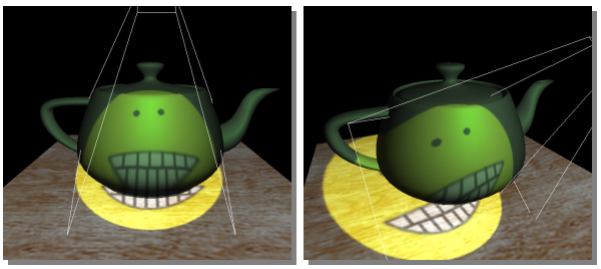


Figura 5.10: Ejemplo de *Projective Texture Mapping*
Fuente: Everitt (2001)

Código 5.3: Código del *fragment shader* para calcular *Projective Texture Mapping*

```
1   // en el vertex shader -> clipSpace = MVP * vec4(FinalPos, 1.0);
2   vec2 normalizedSpace = (clipSpace.xy/clipSpace.w) / 2.0 + 0.5;
3   vec2 refractTextCoord = vec2(normalizedSpace.x, normalizedSpace.y);
4   vec2 reflectTextCoord = vec2(normalizedSpace.x, -normalizedSpace.y);
5
6   vec4 reflectColor = texture(reflectionTexture, reflectTextCoord);
7   vec4 refractColor = texture(refractionTexture, refractTextCoord);
```

Una vez se dispone de las coordenadas de texturas en el correcto espacio se realiza una

mezcla del color de la textura reflejada y refractada que se aplica sobre cada fragmento de la escena, obteniendo como resultado lo que se muestra en la figura 5.11.

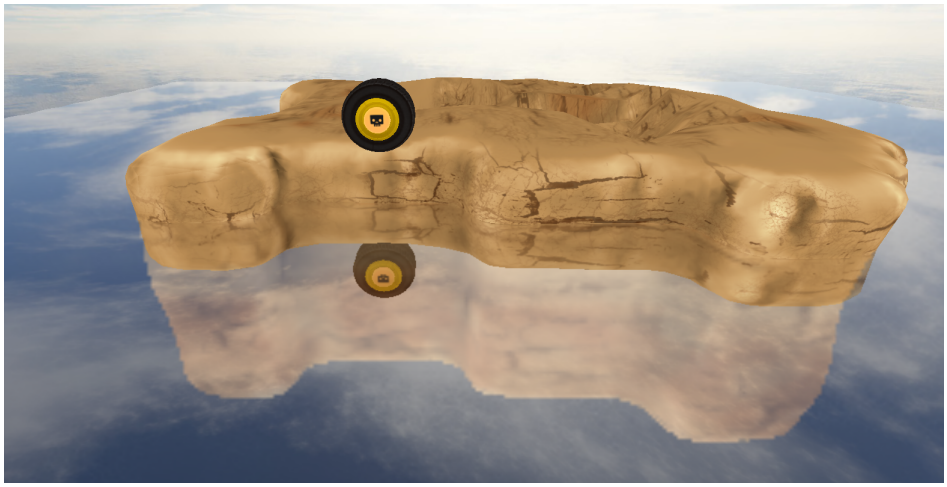
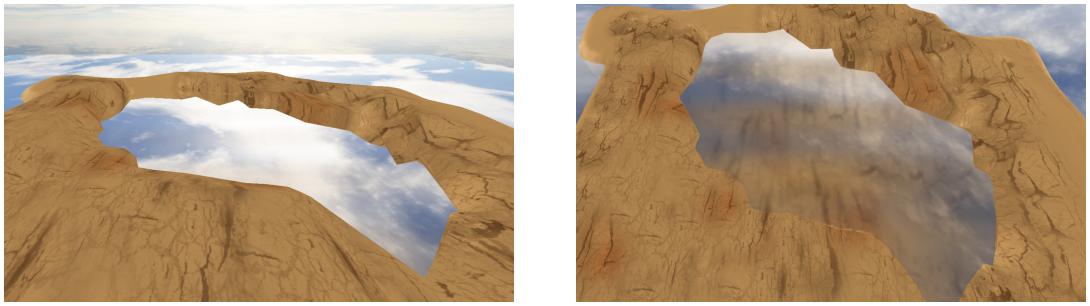


Figura 5.11: Ejemplo resultante de aplicar las texturas de refracción y reflexión

5.4.3. Fresnel Effect

Para calcular el efecto de Fresnel es necesaria la normal del fragmento y el vector de la posición de la cámara al fragmento. Calculando el producto escalar de estos dos vectores se obtiene el ángulo con el que se observa el punto en la superficie del agua, yendo de 0 a 1 si los vectores son más transversales o más paralelos.

Este resultado se utiliza para calcular el color predominante en la mezcla entre las texturas de refracción y reflexión obteniendo como resultado la siguiente figura 5.12. Además, se dispone de un factor de Fresnel que permite modificar lo reflectante que es la superficie del océano.



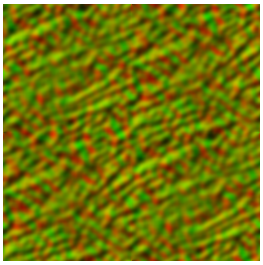
(a) Ángulo resultante grande por lo que destaca más el reflejo

(b) Ángulo resultante pequeño por lo que destaca más la refracción

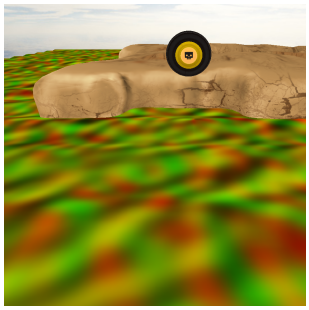
Figura 5.12: Resultado de aplicar el efecto de Fresnel al algoritmo de ondas compuestas con un factor de Fresnel alto para que refleje bastante

5.4.4. Distorsión

Para obtener la distorsión en la textura del océano se envía el *DuDv map* mostrado en la figura 5.13a al *fragment shader*. Esta se aplica junto a las coordenadas de textura de la malla del océano obteniendo el resultado que se muestra en la figura 5.13b, extrayendo para cada fragmento los canales de color R y G de la textura en esa posición. Luego, se transforman los valores obtenidos del espacio de color (0, 1) al (-1, 1) para que la distorsión se pueda aplicar en cualquier dirección. Además, se multiplican los valores de desplazamiento resultantes por un factor de fuerza para poder variar lo brusca que es la distorsión aplicada, añadiendo una variable que permite controlar la escala de la textura de distorsión.



(a) *DuDv map* utilizado en el proyecto
Fuente: ThinMatrix (2015)



(b) Mapa de distorsión aplicado sobre el océano

Figura 5.13: Se muestra el mapa de distorsión aplicado con *Projective Texture Mapping* sobre la malla del agua para comprobar su correcto funcionamiento

Tras esto, se aplica el vector de dos dimensiones resultante a las coordenadas de las texturas de refracción y reflexión calculadas en el apartado 5.4.2. Debido a que las coordenadas resultantes pueden sobrepasar los bordes de pantalla se limitan a un valor mínimo de 0 y uno máximo de 1 para evitar que se generen defectos gráficos en estos. El resultado obtenido se muestra en la siguiente figura 5.14.

Figura 5.14: Resultado de aplicar distorsión pequeña al agua

Por último, se va a dotar de movimiento a esta textura para que no permanezca estática aplicándole un valor de desplazamiento dependiente del tiempo transcurrido en la dirección

deseada. En el siguiente código 5.4 se observa su cálculo en el *fragment shader*.

Código 5.4: Código del *fragment shader* para calcular distorsión en el agua

```
vec2 distortedTextureCoords = vec2(TexCoords.x, TexCoords.y + moveFactor);
vec2 distorsionDuDvMap = (texture(dudvMap, distortedTextureCoords).rg * 2.0
    ↪ - 1.0) * distortionStength;

refractTextCoord += distorsionDuDvMap;
// clampeamos para evitar el bug en los limites de la pantalla
refractTextCoord = clamp(refractTextCoord, 0.001, 0.999);

reflectTextCoord += distorsionDuDvMap;
// clampeamos para evitar el bug en los limites de la pantalla
reflectTextCoord.x = clamp(reflectTextCoord.x, 0.001, 0.999);
reflectTextCoord.y = clamp(reflectTextCoord.y, -0.999, -0.001);

vec4 reflectColor = texture(reflectionTexture, reflectTextCoord);
vec4 refractColor = texture(refractionTexture, refractTextCoord);
FragColor = mix(reflectColor, refractColor, frestnelFactor);
```

5.4.5. Mapeado de normales (*Normal Mapping*)

Para implementar el mapeado de normales se envía al *fragment shader* el *normal map* (Figura 5.15), se utilizan las coordenadas de textura que tiene el fragmento para extraer el vector de tres dimensiones definido por el color. El valor obtenido se pasa del rango (0, 1) al (-1, 1), ya que se pueden tener normales que apunten en la dirección negativa de un eje. Tras esto, surge un problema con la orientación de los vértices, debido a que el mapa de normales devuelve la normal que se aplica directamente sobre un plano orientado en su misma dirección, es decir, con las normales orientadas en la dirección positiva del eje Y. Por lo que si este vértice está orientado en una dirección distinta, la textura del mapa de normales va a permanecer orientada en el mismo eje de coordenadas, entonces, al aplicarse sobre este vértice proporcionará un resultado incorrecto al disponer de un sistema de coordenadas diferente.

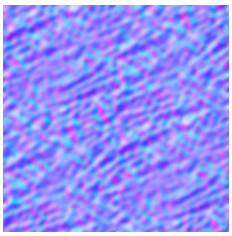


Figura 5.15: Mapa de normales utilizado en el proyecto
Fuente: ThinMatrix (2015)

Para orientar el mapa de normales en la dirección que tiene el vértice va a ser necesaria calcular una matriz de transformación con la que modificar el espacio en el que se encuentra este mapa. Para ello, se utiliza la normal de este vértice para calcular la tangente y bitangente como se puede ver en de Vries (2020, Cap.37). La librería Assimp proporciona el resultado del

cálculo de la tangente y bitangente cuando importamos un modelado para que solo haya que generar la matriz, sin embargo, como en la malla del océano se modifican las orientaciones de los vértices dinámicamente desde la *GPU* no sirve este precálculo. Por lo que estas operaciones se van a mover al *fragment shader* realizando el cálculo con la normal del fragmento explicado extendidamente en Schüler (2013).

Una vez calculada la matriz de transformación se aplica sobre las normales del mapa para obtener las normales en el correcto sistema de coordenadas. Ahora, se hace uso de estas para calcular la luz especular con Phong como se muestra en el código 5.5 para obtener unas olas pequeñas cuando la luz incide sobre la malla. A la que se le añade una tonalidad azul para mejorar el resultado visual, además se le agrega la variable **reflectivity** para controlar su brillo.

Código 5.5: *Fragment shader* para calcular la luz especular con la normal resultante obtenida del mapa de normales

```
1   vec3 PerturbedNormal = calculatePerturbNnormal(norm, viewDir, ↩
        ↪ distortedTextureCoords);
2
3   vec3 lightDir = normalize(pointLights[0].position - FragPos);
4   vec3 reflectDir = reflect(-lightDir, PerturbedNormal); //Angulo reflectado
5   float spec = 0.0;
6   vec3 specular = vec3(0.0,0.0,0.0);
7   spec = pow(max(dot(viewDir, reflectDir), 0.0), shineWave); //Formula de la ↩
        ↪ luz especular
8   specular = pointLights[0].specular * spec * reflectivity; //Multiplicamos ↩
        ↪ todo
9
10  FragColor = mix(reflectColor, refractColor, frestnelFactor);
11  FragColor = mix(FragColor, vec4(0.0, 0.3, 0.5, 1.0), 0.3) + vec4(specular, ↩
        ↪ 0.0);
```

Por último, estas olas pequeñas también deben tener un desplazamiento, por lo que se aplica el mismo *offset* a las coordenadas de textura que se aplicaba al calcular la distorsión para que se desplacen en la misma dirección. Obteniendo como resultado la figura 5.16.

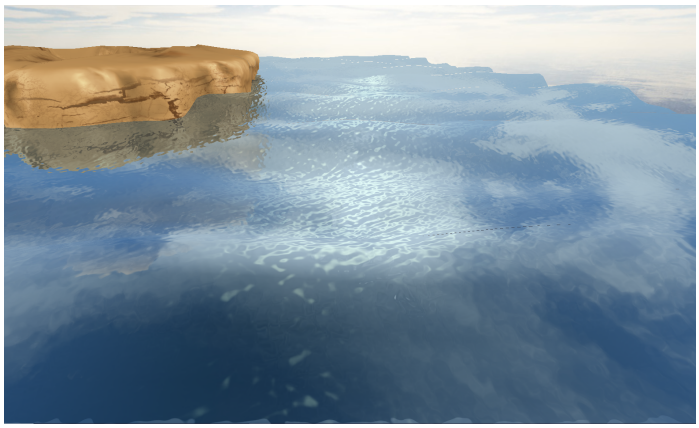


Figura 5.16: Resultado de aplicar el mapeado de normales al agua junto a las técnicas de iluminación explicadas anteriormente

5.4.6. Profundidad del agua

Para aplicar el efecto de profundidad al agua es necesario calcular el *depth buffer*, para ello OpenGL tiene una serie de directivas que permiten almacenar el *buffer* en una textura al realizar el *renderizado* de la escena. Por lo que se va a aprovechar el proceso de *renderizado* con el que se calcula la textura refractada para obtener y almacenar también el *buffer* de profundidad de los elementos que la componen en una textura, y luego enviarla al proceso final de dibujado utilizándola en el *fragment shader*. Los datos de esta textura se encuentran en el canal rojo, ya que es un mapa en escala de grises que se oscurece al aproximarse un elemento a la cámara.

Una vez en el *fragment shader* del océano, se calcula la profundidad del agua utilizando la distancia a los elementos de la escena que se obtiene de la textura y la distancia a la superficie del agua. OpenGL define una serie de variables que aportan valores del estado actual del *renderizado*, la variable **gl_FragCoord** justo proporciona la distancia que hay de la cámara al elemento que se está *renderizando*. Con la diferencia entre estas ya se dispondría de la profundidad del agua en cada fragmento de la escena, sin embargo, los *depth buffer* de OpenGL no siguen una relación lineal para su distancia (de Vries, 2020, Cap.22), por lo que se usa la fórmula 5.10 en las dos distancias anteriores para modificarlas a un valor correcto antes de restarlas. Las variables *near* y *far* hacen referencia a los planos lejano y cercano de la cámara escogidos para calcular la matriz de proyección. El resultado de la resta se divide por un valor que permite controlar la distancia a la que comienzan a interpretarse los elementos como lejanos o cercanos y se limita a un rango de (0, 1) obteniendo los valores que se muestran en la figura 5.17.

$$RealDepth = 2.0 * near * far/(far + near - (2.0 * depth - 1.0) * (far - near)) \quad (5.10)$$

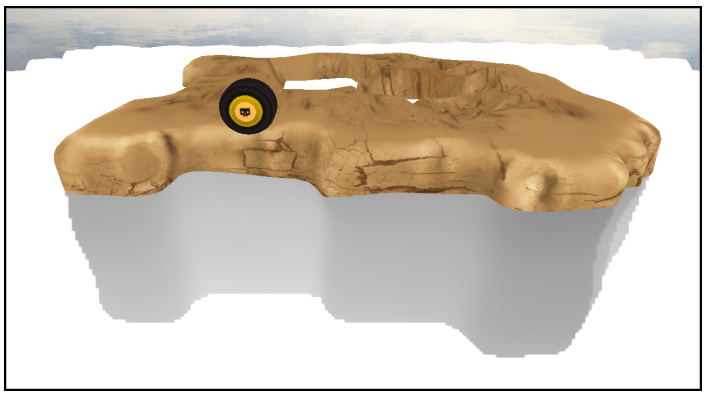
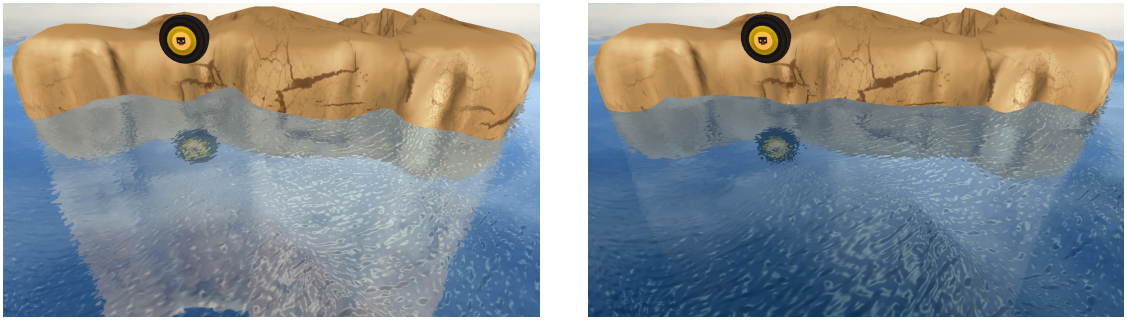


Figura 5.17: Se muestra el *depth buffer* calculado, aplicado sobre la superficie del océano

A continuación, se va a oscurecer la textura refractada según la profundidad, para ello se le resta a uno el valor resultante de la profundidad para invertirlo y que sea el color claro el que represente los elementos poco profundos. De esta manera, se multiplica en cada fragmento este valor resultante por el color de la textura refractada obteniendo el resultado deseado observado en la figura 5.18.



(a) Profundidad del agua desactivada **(b)** Profundidad del agua activada

Figura 5.18: Comparativa de aplicar el oscurecimiento por profundidad al algoritmo de Gerstner junto con las características anteriores de iluminación

5.4.7. Cáustica del agua

Para el desarrollo de la cáustica del agua se ha implementado el método mencionado en el apartado 3.2.4 (Copiado de texturas), enviándose la textura mostrada en la figura 5.19 al proceso de *renderizado* que calcula la textura refractada. La textura cáustica se aplica a los fragmentos de los modelados que se encuentran por debajo de la superficie del agua. Para ello, se dispone del plano de *clipping* utilizado en el cálculo de la refracción con el que se conoce la altura del océano.

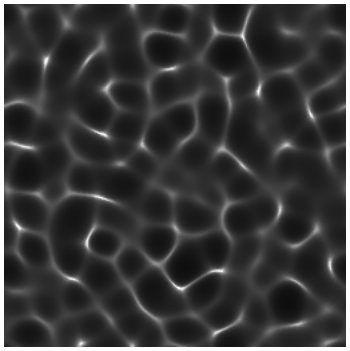
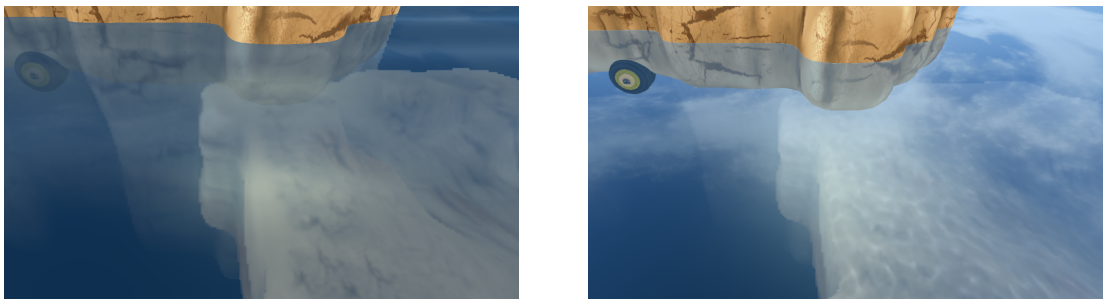


Figura 5.19: Textura cáustica utilizada en el proyecto
Fuente: Dicker (2012)

En el *fragment shader* se obtienen con los datos anteriores los fragmentos sobre los que hay que aplicar esta textura. Además, se va a reducir la intensidad de este efecto en los fragmentos en los que el sol no incide directamente mediante el cálculo del producto escalar entre la normal del fragmento y el vector del fragmento a la fuente de iluminación. Por último, se aplican dos texturas, en lugar de una, con movimiento en direcciones diferentes para obtener un mejor resultado al diferenciarse menos el patrón de la textura. En la figura 5.20 se aprecia el efecto de aplicar esta técnica.



(a) Sin efecto cáustico **(b)** Con efecto cáustico

Figura 5.20: Comparativa de aplicar el efecto cáustico

5.5. Diseño de la interfaz

Para el desarrollo de la interfaz se han utilizado las librerías ImGUI y ImPlot buscando presentar todas las funcionalidades configurables desarrolladas en el proyecto, para poder apreciar adecuadamente las características de las que disponiese un desarrollador si utilizase este motor gráfico. También, se muestran estadísticas sobre la aplicación que son esenciales para realizar un análisis de datos y extraer conclusiones, además de facilitar algunas tareas de depuración. A continuación, se muestran todos los apartados con valores configurables:

- **Algoritmos de oleaje** (Figura 5.21): Se selecciona uno de los algoritmos implementados que se desea observar, mostrando un contador con el número de vértices de la malla. Se puede modificar la resolución y longitud de la malla pulsando el botón para regenerarla y que se apliquen los valores seleccionados.

  En el apartado de configuración del algoritmo se muestra la configuración específica que tiene cada algoritmo de oleaje. En el caso del de **Ondas compuestas** se puede seleccionar el número de ondas y definir la amplitud, velocidad, longitud de onda y dirección de cada una. Para **Gerstner** se pueden modificar las mismas propiedades que en el anterior, pero quitando la velocidad y añadiendo el desplazamiento de las olas. En **Tessendorf** es modificable el número de secciones, la longitud de onda, la dirección y el parámetro de Philips.

- **Iluminación** (Figura 5.22): Se puede seleccionar la posición de la fuente de luz, las propiedades de iluminación que afectan a los materiales de la escena y la atenuación con la distancia.

- ***Shader*** (Figura 5.23): Se pueden configurar las propiedades del material de la malla del agua que influirá en el cálculo de la iluminación de Phong. También, se pueden modificar los valores de las texturas utilizadas para aplicar el mapeado de normales y la distorsión. Pudiendo variar el *tileado* de las texturas, la velocidad con las que se desplazan, su dirección de movimmiento, la fuerza de distorsión aplicada sobre el agua que tendrán los reflejos, y el rango de profundidad el agua visible. Además, se pueden activar y desactivar las funcionalidades de iluminación desarrolladas.

- **Estadísticas** (Figura 5.24): Se muestran contadores y gráficas con el *framerate* actual de la aplicación y el tiempo que tarda en procesarse cada ciclo de juego.

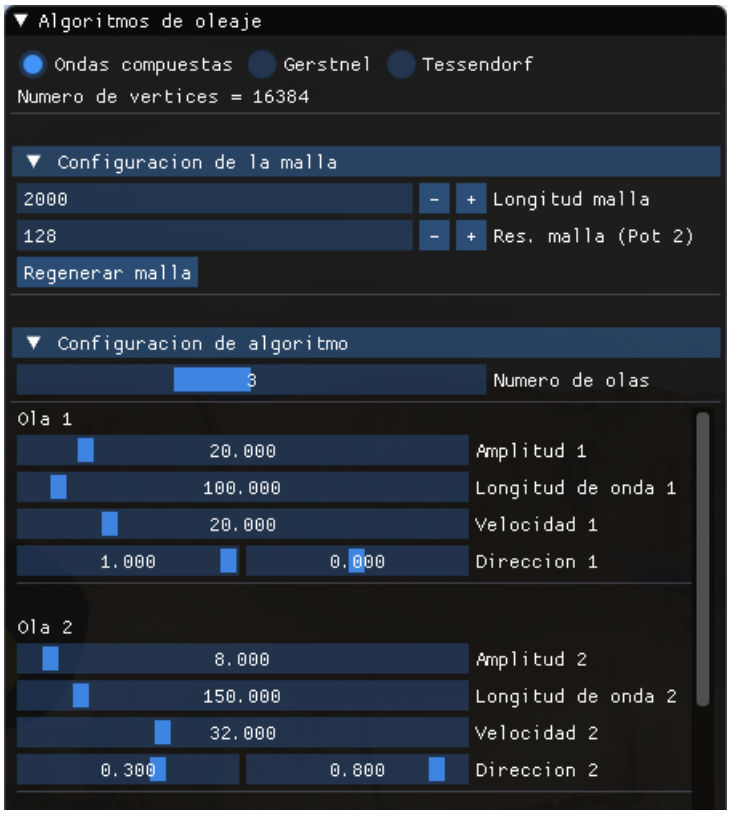


Figura 5.21: Interfaz de algoritmos de oleaje

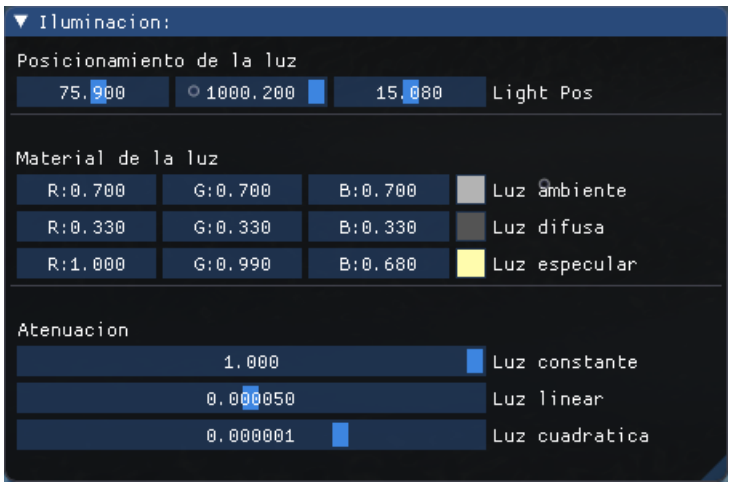


Figura 5.22: Interfaz de la iluminación de la escena

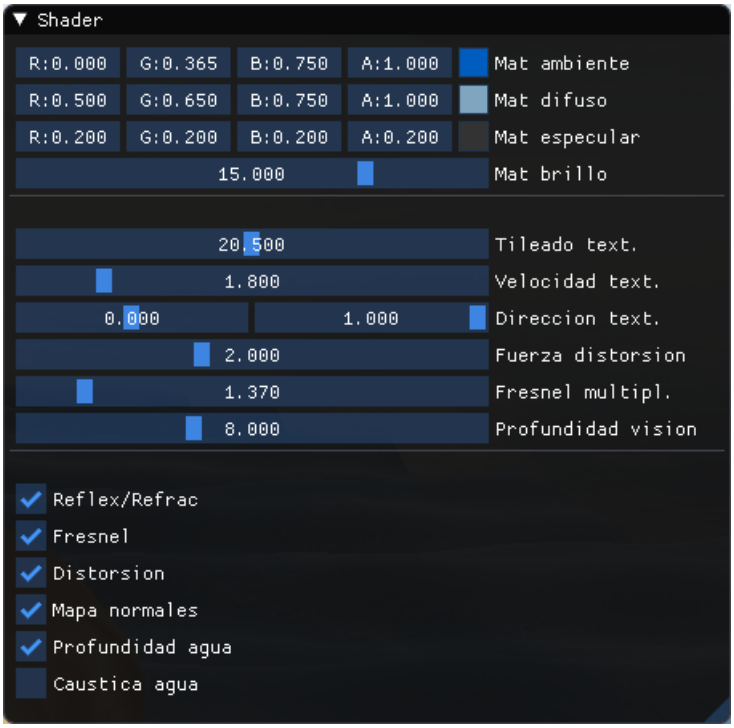


Figura 5.23: Interfaz de configuración de los *shaders*

Figura 5.24: Interfaz de estadísticas de la aplicación

5.6. Pruebas y validación

La planificación seguida para comprobar el correcto funcionamiento de las funcionalidades desarrolladas ha sido realizar diversas pruebas una vez finalizada cada tarea modificando los valores configurables para combrobar que el resultado es el adecuado. Una vez finalizado el *sprint* se comprueba que no se genera ningún problema al unificarla con el resto de las funcionalidades. Además, el uso de variables editables desde la interfaz con el uso de ImGUI facilita el proceso de validación al no tener que volver a compilar para observar el resultado tras modificar una variable.

La herramienta principal utilizada para solventar los errores encontrados ha sido el depurador que incorpora MVS. Que permite establecer puntos de parada en los que pausar la ejecución del programa y comprobar los valores de todas las variables y estructuras accesibles en ese entorno. Además, su *intellisense*[3] permite mostrar visualmente errores básicos de sintaxis en C++ mientras se programa para poder solucionarlos antes de compilar y así agilizar el proceso de desarrollo ahorrando tiempo.

Para los *shaders* se imprimen por terminal los mensajes en caso de que surjan errores de compilación para poder solucionarlos. El proceso de depuración de su código es más costoso al tratarse de archivos externos a Visual Studio que se compilan en tiempo de ejecución, por lo que se ha desarrollado un *geometry shader* que permite mostrar las normales de los vértices y las caras de los triángulos de la malla para comprobar su correcta orientación como se ve en la figura 5.25. Este *shader* se ha desarrollado para depurar principalmente problemas de las orientaciones de las normales en los vértices al calcular los algoritmos de oleaje.

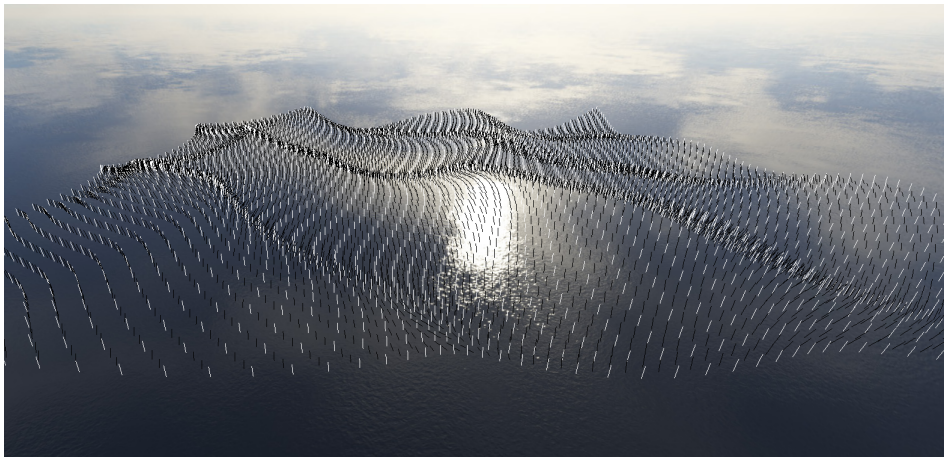


Figura 5.25: Se aplica el *Geometry Shader* de depuración a la malla generada en Gerstner para observar la orientación de las normales

[3]Función que incorpora MVS que colorea el texto viendo más clara la sintaxis y remarcando los posibles fallos de programación en el lenguaje empleado. También, incorpora sugerencias y funcionalidades de autocompletado.

6. Análisis y resultados

El número medio de vértices que se utiliza en la mayoría de los análisis es 90 mil, debido a que es la media de vértices visibles en pantalla que tiene un océano con buena definición gráfica en un videojuego, tras emplear algoritmos que descartan vértices que se encuentran fuera de la cámara y alejados de esta. Los siguientes análisis se han realizado con una resolución de la aplicación de 1920x1080 píxeles, en modo ventana y las características del ordenador descritas en el apartado 2.1. El rendimiento medio de la aplicación del que se parte sin utilizar ningún algoritmo de oleaje ni ninguna de las técnicas gráficas desarrolladas es de unos 185 *FPS* o 5,45 milisegundos por ciclo de juego.

En los algoritmos de Ondas compuestas y Gerstner se han conformado con 4 ondas simples al observarse que con este valor se obtienen buenos resultados visuales sin notarse demasiado la repetición, por lo que su coste se incrementa al calcularse 4 veces por vértice. Lo que se observa en la figura 6.1 es que el coste de Tessendorf similar a los otros dos algoritmos, aunque este solo se calcule una vez. Esto se debe a que el cálculo de Tessendorf tiene un coste de $n \log n$ mientras que las Ondas compuestas y Gerstner tienen un coste lineal (n). Por otra parte, Gerstner tiene un coste un poco superior al algoritmo de Ondas compuestas al debido a que se añaden las operaciones que realizan el cálculo del desplazamiento.

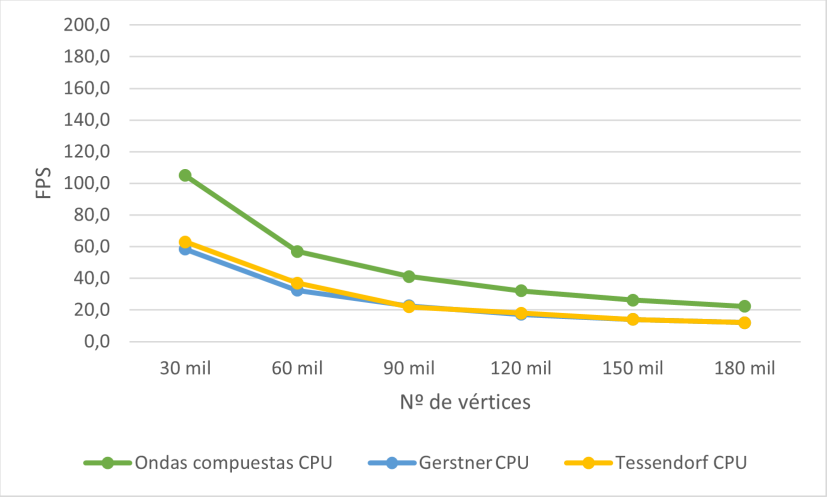
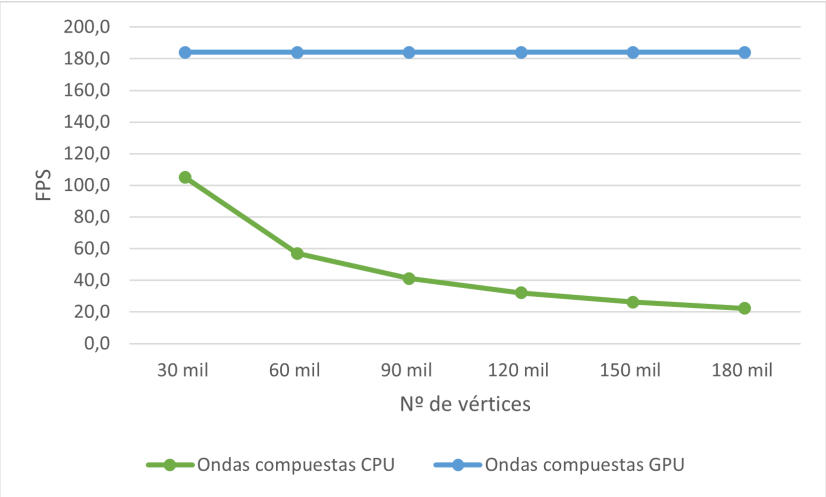
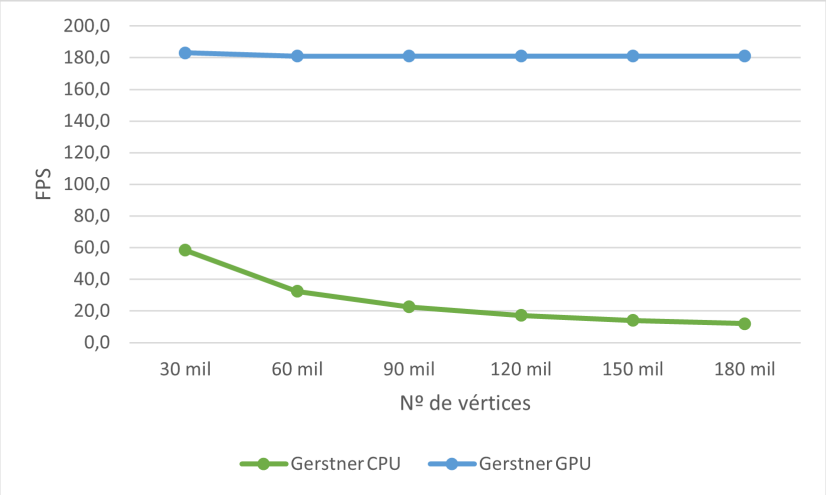


Figura 6.1: Gráfica que muestra el rendimiento en *FPS* al aplicar los algoritmos de oleaje en la *CPU*

En la figura 6.2 se observa el gran ahorro de rendimiento obtenido al pasar los algoritmos a de la *CPU* a la *GPU*, esto es debido a la capacidad de la gráfica a la hora de paralelizar los cálculos como se ha comentado en el apartado 3.3.3. Por lo que, el coste de llevar los datos a la gráfica para calcular estos algoritmos es beneficioso al disponer de un gran aumento en el rendimiento.



(a) Algoritmo de Ondas compuestas



(b) Algoritmo de Gerstner

Figura 6.2: Gráficas que muestran la comparativa en *FPS* de implementar los algoritmos de la *CPU* en la *GPU*

Para Tessendorf se ha desarrolado el sistema que calcula el algoritmo solo en una pequeña cantidad de vértices y envía copias de esa sección desplazadas para formar el océano. Se ha definido la sección a calcular de unos 16 mil vértices, que al duplicarla es difícil apreciar un patrón repetitivo. Con lo mostrado en la figura 6.3 no variar prácticamente el *framerate*, por lo que se deduce que el auténtico coste se encuentra en el cálculo del algoritmo y no en el número de vértices que se envían a la gráfica.

En el análisis realizado en la figura 6.4 se han introducido en la escena varios modelados, con un número total de 50 mil vértices, para apreciar cuáles de las técnicas de iluminación tienen un mayor consumo. Se observa el gran consumo de los reflejos y refracciones, esto se debe a que la escena se *renderiza* dos veces adicionales para calcular esta técnica de iluminación. En el ordenador donde se han realizado los análisis estas técnicas conjuntamente tardan en

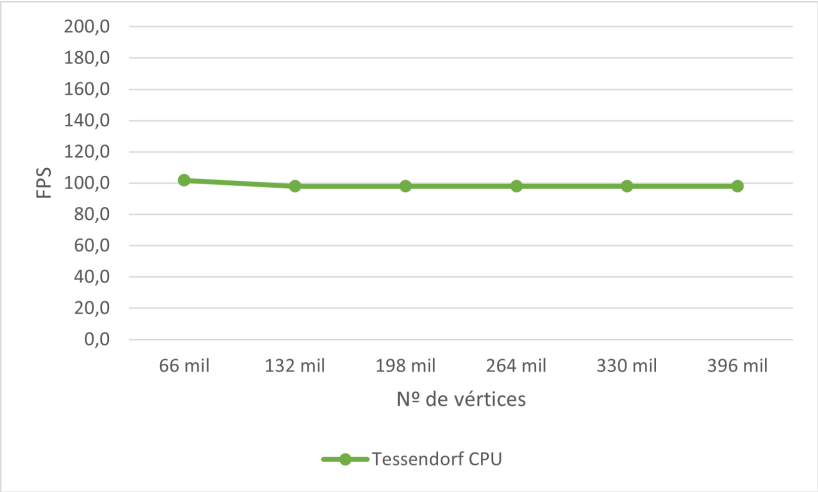


Figura 6.3: Gráfica que muestra el rendimiento en *FPS* de Tessendorf al solo aplicar el cálculo del algoritmo a una sección de la malla

calcularse unos 0,47 milisegundos, consumiendo un rendimiento de 1,8 *FPS*. Se observa que el coste que lleva calcular estas técnicas es prácticamente despreciable en comparación con el coste de calcular Tessendorf.

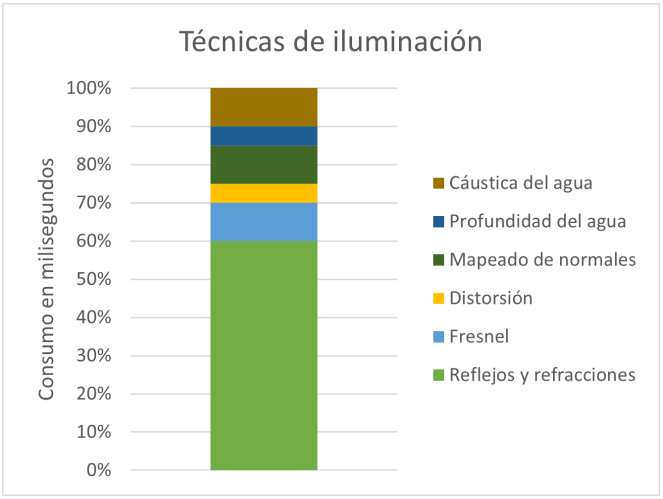


Figura 6.4: Gráfica que muestra el porcentaje que conlleva calcular cada una de las técnicas de iluminación

Por último, se muestra en la figura 6.5 el rendimiento tras aplicar las técnicas de iluminación junto a los algoritmos de oleaje. Los algoritmos de Ondas compuestas o Gerstner se podrían utilizar en cualquier videojuego, sin embargo, el de Tessendorf habría que tratar de mejorar su rendimiento en caso de querer utilizarlo en una gran superficie de agua con mucha definición de vértices. En el anexo E se muestran todas las gráficas con el coste temporal en milisegundos en lugar de en número de *FPS*.

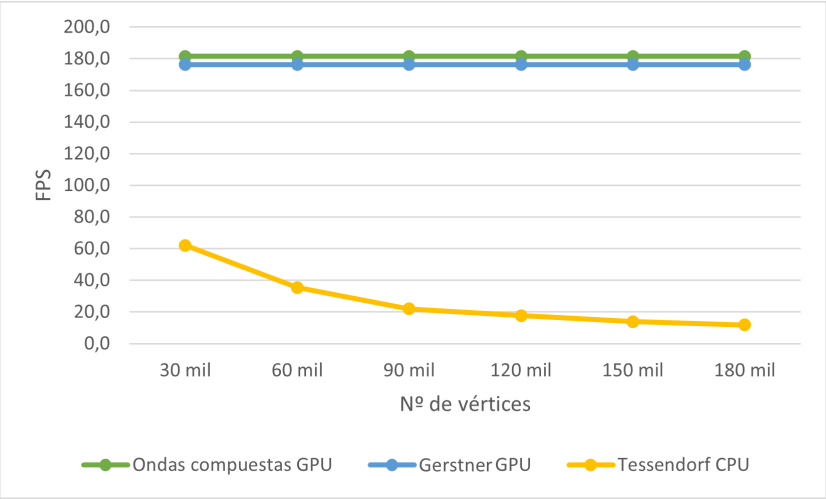


Figura 6.5: Gráfica que muestra el coste final en *FPS* de los algoritmos junto a las técnicas de iluminación

6.1. Resultados gráficos

En las figuras 6.6, 6.7 y 6.8 se muestran los resultados visuales que se han obtenido en el TFG para cada algoritmo de oleaje, utilizando todas las técnicas de iluminación desarrolladas.

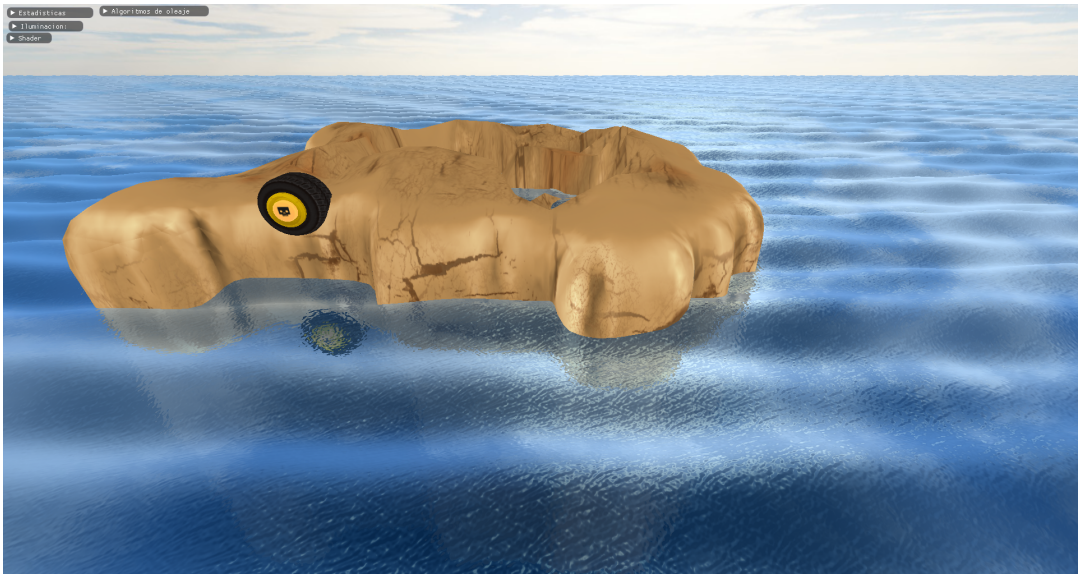


Figura 6.6: Resultado gráfico final del algoritmo de Ondas compuestas

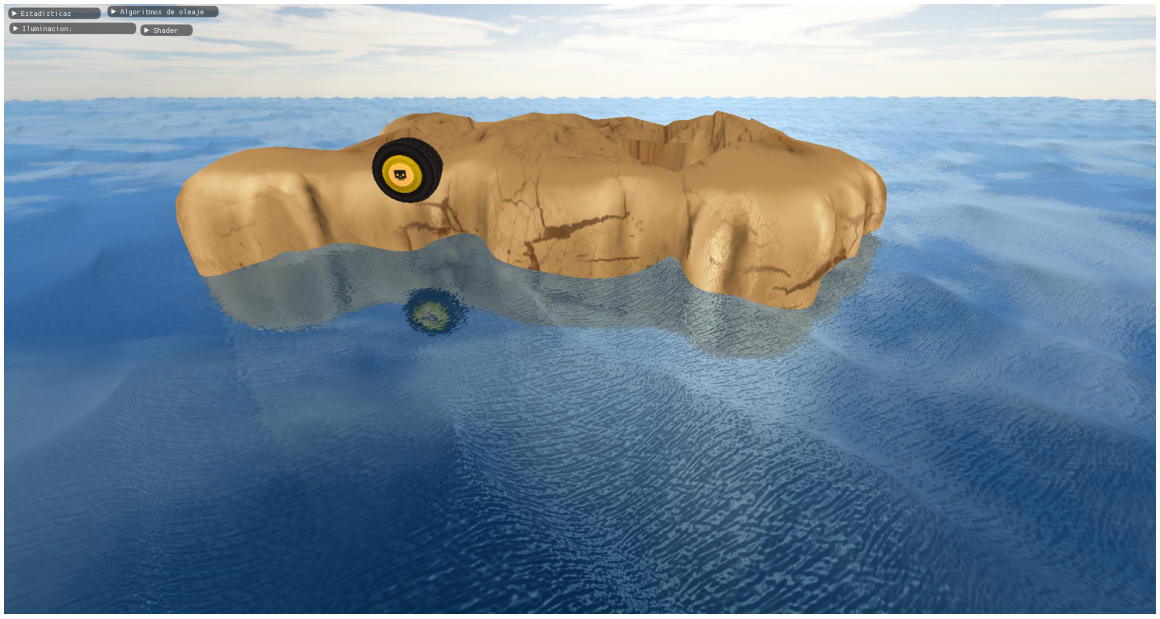


Figura 6.7: Resultado gráfico final del algoritmo de Gerstner

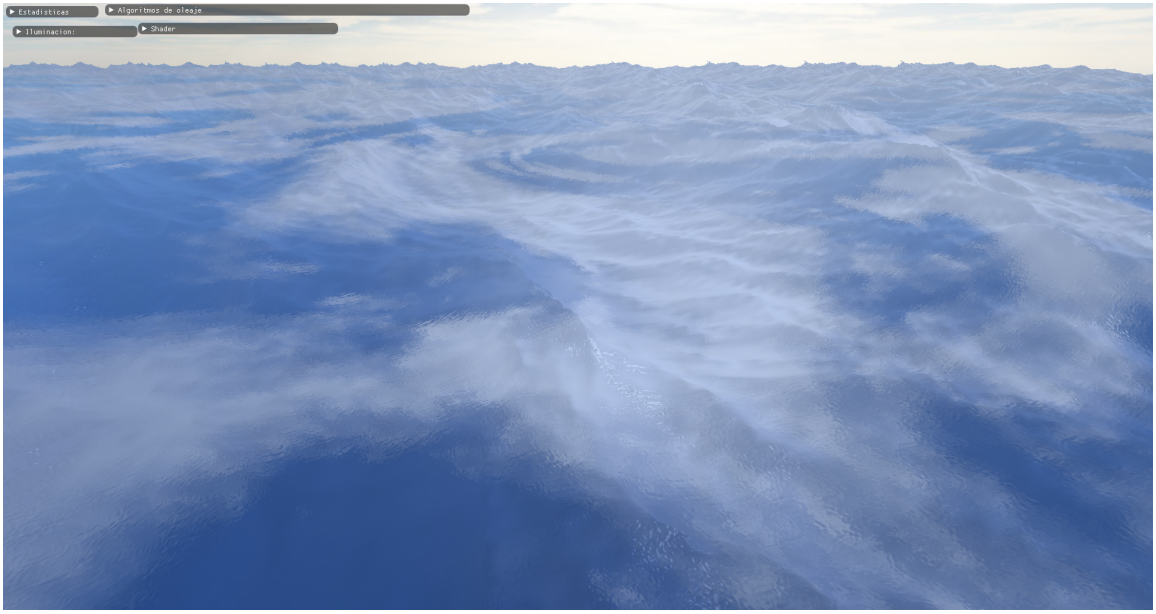


Figura 6.8: Resultado gráfico final del algoritmo de Tessendorf

7. Conclusiones y trabajo futuro

7.1. Conclusiones

En el trabajo se ha conseguido obtener una visión general de cómo se desarrollan los océanos en videojuegos, tras estudiar en el estado de la cuestión algunos de los algoritmos de oleaje, métodos de optimización y técnicas de iluminación más comunes. Tras esto, se han seleccionado los algoritmos de oleaje de Ondas compuestas, Gerstner y Tessendorf, al ser los más utilizados en este campo y con los que se obtiene un mejor resultado para un océano, sorprendiéndome la cantidad de videojuegos que hacen uso del algoritmo de Tessendorf incorporándole pequeñas modificaciones. Luego, se ha seleccionado el método de pasar los cálculos a la gráfica como el método de optimización a desarrollar. Por último, se han seleccionado algunas técnicas de iluminación que dotarían al trabajo de un buen resultado visual.

A continuación, se ha adaptado el motor gráfico a la plataforma de Windows para que se pudiese comenzar a trabajar en este. Seguidamente, se ha desarrollado un generador de mallas que ha servido como base para acabar implementando los algoritmos especificados; Siendo el de Tessendorf más complejo de lo esperado, pero obteniendo unos resultados bastante espectaculares. Los cálculos de los algoritmos de Ondas compuestas y Gerstner se han optimizado moviéndose del procesador a la gráfica. El de Tessendorf no se ha movido debido a su complejidad, ya que, en comparación con los otros dos algoritmos, el cálculo del algoritmo de Tessendorf en un vértice depende de los otros, por lo que habría que realizar procesos más complejos generando previamente texturas en la *CPU* para pasar la información a la *GPU* y así calcular el resto del algoritmo. Luego, tras agregar todas las técnicas de iluminación seleccionadas principalmente en los *shaders*, se ha añadido una interfaz gráfica con las características modificables que se han especificado. La técnica del mapeado de normales ha sido la que más me ha impresionado con la gran calidad visual que aporta al combinarse con el resto y a la variedad de materiales, no necesariamente líquidos, donde es aplicable.

Finalmente, se ha realizado un análisis de todas las funcionalidades desarrolladas, donde me ha sorprendido la inmensa eficiencia obtenida al pasar el cálculo de los algoritmos a la gráfica siendo el coste de Ondas Compuestas y Gerstner menor del esperado, pudiendo ser aplicables a cualquier videojuego sin necesidad de aplicar alguna técnica de optimización adicional. Mientras, el rendimiento obtenido en el algoritmo de Tessendorf no es suficiente para aplicarlo a grandes superficies de agua. En cuanto a las técnicas de iluminación, el porcentaje de consumo que tiene cada una era el previsto al relacionarse directamente con la cantidad de procedimientos que requieren para obtenerse.

Concluyendo, en este trabajo he aprendido bastante acerca de un campo que antes desconocía completamente y ahora me apasiona. He podido pulir mis conocimientos sobre el uso de las metodologías en trabajos a largo plazo que aprendí en el proyecto grupal realizado durante el cuarto curso de la titulación. Además, de mejorar otras características como el nivel de programación o la capacidad de investigación y autoaprendizaje que me servirán

para los próximos proyectos que realice.

7.2. Trabajo futuro

Debido a la gran extensión de la que dispone la temática tratada en este trabajo, hay una serie de características que se podrían desarrollar o algunas mejoras que surgieron mientras se desarrollaba el proyecto, por lo que se van a comentar una serie de mejoras con las que se podría continuar este trabajo.

En cuanto a los algoritmos de oleaje, se podría tratar de mover los cálculos de Tessendorf del procesador a la gráfica, para así obtener una mejora sustancial en el rendimiento. Además, de tratar de desarrollar la técnica de particionado del espacio con la que se podría reducir el número de vértices procesados a más de la mitad si el jugador se encontrase en un punto intermedio de la malla.

En el apartado visual, la técnica que genera espuma en el oleaje le aportaría un gran aumento en el realismo visual, siendo esta mejora superior a la que han aportado otras ya desarrolladas, cómo la cáustica del agua que no es tan influyente en un océano. Aplicar sombras de los objetos sobre el agua ha sido otra característica que ha surgido durante el desarrollo, dispone de una implementación similar a las técnicas de la profundidad o a los reflejos y refracciones. Finalmente, se podría tratar de aplicar la técnica de *Path Tracing* al renderizado del agua para obtener una mejora sustancial en la iluminación resultante.

Bibliografía

Alisavakis, H. (2019, septiembre). *My take on shaders: Water Shader.* Descargado de https://halisavakis.com/my-take-on-shaders-water-shader/

Ang, N., Catling, A., Ciardi, F. C., y Kozin, V. (2018). The technical art of sea of thieves. En *Acm siggraph 2018 talks.* New York, NY, USA: Association for Computing Machinery. Descargado de https://doi.org/10.1145/3214745.3214820 doi: 10.1145/3214745.3214820

Bruneton, E., Neyret, F., y Holzschuch, N. (2010, mayo). Real-time Realistic Ocean Lighting using Seamless Transitions from Geometry to BRDF. *Computer Graphics Forum*, *29*(2), 487–496. Descargado 2020-11-15, de http://doi.wiley.com/10.1111/j.1467-8659.2009.01618.x doi: 10.1111/j.1467-8659.2009.01618.x

de Boer, W. (2000, 01). Fast terrain rendering using geometrical mipmapping.

de Vries, J. (2014, junio). *LearnOpenGL - Normal Mapping.* Descargado de https://learnopengl.com/Advanced-Lighting/Normal-Mapping

de Vries, J. (2020). *Learn OpenGL: Learn modern OpenGL graphics programming in a step-by-step fashion.* (OCLC: 1198376019)

Dicker, L. (2012, noviembre). *Water Caustics Effect.* Descargado de https://opengameart.org/content/water-caustics-effect-small

Dunn, I., y Wood, Z. (2016, diciembre). *Chapter 2: The Graphics Pipeline.* Descargado de https://graphicscompendium.com/intro/01-graphics-pipeline

Elmore, W. C., y Heald, M. A. (1985). *Physics of waves.* New York: Dover Publications.

Everitt, C. (2001). Projective texture mapping..

Federico Balmaceda, D. (2015, septiembre). *GeoGebra - Superposición de ondas.* Descargado de https://www.geogebra.org/m/cMQpahe2

Fernando, R. (Ed.). (2004). *GPU gems: programming techniques, tips, and tricks for real-time graphics.* Boston: Addison-Wesley.

Gentile, N. (2019, agosto). *Lo que NO te contaron sobre Ray Tracing.* Descargado de https://www.youtube.com/watch?v=tbsudki8Sro

Green, S. (2010, marzo). *Screen Space Fluid Rendering for Games (Nvidia).* Moscone Center (San Francisco, CA). Descargado de https://developer.download.nvidia.com/presentations/2010/gdc/Direct3D_Effects.pdf

Harding, S. (2021, junio). *What Is Nvidia DLSS? A Basic Definition.* Descargado de https://www.tomshardware.com/reference/what-is-nvidia-dlss

Johanson, C. (2004). *Real-time water rendering: Introducing the projected grid concept* (Tesis Doctoral, Lund University). Descargado de https://fileadmin.cs.lth.se/graphics/theses/projects/projgrid/projgrid-hq.pdf

Kershaw Winder, R. (2000, diciembre). *The Kinetic PR Quadtree (University of Maryland).* Descargado de https://www.cs.umd.edu/~mount/Indep/Ransom/index.htm

Khan, J. (2013, septiembre). *TerraMonkey - The jMonkeyEngine Terrain System.* Descargado de https://wiki.jmonkeyengine.org/docs/3.4/core/terrain/terrain.html#geo-mip-mapping

Li, K., y Wu, L. (2007). *Water Simulating in Computer Graphics* (Tesis Doctoral, Växjö University, School of Mathematics and Systems Engineering). Descargado de https://www.diva-portal.org/smash/get/diva2:205412/FULLTEXT01.pdf

Möller, T. (2018). *Real-time rendering* (Fourth edition ed.). Boca Raton: Taylor & Francis, CRC Press.

Müller, M., Charypar, D., y Gross, M. (2003, 07). Particle-based fluid simulation for interactive applications. En (Vol. 2003, p. 154-159).

Ohorodnichuk, M. (2018, octubre). *Cliffs of the desert Free low-poly 3D model.* Descargado de https://www.cgtrader.com/free-3d-models/various/various-models/cliffs-of-the-desert

Rojas, J. (2013, agosto). Getting Started with Videogame Development. En *2013 26th Conference on Graphics, Patterns and Images Tutorials* (pp. 1–5). Arequipa, Peru: IEEE. Descargado 2021-06-12, de http://ieeexplore.ieee.org/document/6949393/ doi: 10.1109/SIBGRAPI-T.2013.10

Salmon, R. (2021, March). *Introduction to ocean waves.* Descargado 03/05/2021, de http://pordlabs.ucsd.edu/rsalmon/111.textbook.pdf

Schüler, C. (2013, enero). *Followup: Normal Mapping Without Precomputed Tangents | The Tenth Planet.* Descargado de http://www.thetenthplanet.de/archives/1180

Stuhlmeier, R. (2015, 07). Gerstner's water wave and mass transport. *Journal of Mathematical Fluid Mechanics*, *17*, 8. doi: 10.1007/s00021-015-0219-4

Sweeney, T. (2020, diciembre). *Path Tracer.* Descargado de https://docs.unrealengine.com/4.26/en-US/RenderingAndGraphics/RayTracing/PathTracer/

Tessendorf, J. (2001, 01). Simulating ocean water. *SIG-GRAPH'99 Course Note.*

ThinMatrix. (2015, julio). *Textures.* Descargado de https://www.dropbox.com/sh/z21kj5s448vzfql/AADy_KocZ6gDgK1lYjv7_SBAa?dl=0

Wallace, E. (2016, enero). *Rendering Realtime Caustics in WebGL.* Descargado 2020-09-09, de `https://medium.com/@evanwallace/rendering-realtime-caustics-in-webgl-2a99a29a0b2c`

Walt Disney Animation Studios. (2016, septiembre). *Disney's Practical Guide to Path Tracing.* Descargado de `https://www.youtube.com/watch?v=frLwRLS_ZR0`

Yu, Q., Neyret, F., Bruneton, E., y Holzschuch, N. (2009, abril). Scalable Real-Time Animation of Rivers. *Computer Graphics Forum*, *28*(2), 239-248. Descargado de `https://hal.inria.fr/inria-00345903` doi: 10.1111/j.1467-8659.2009.01363.x

Zucconi, A. (2019, septiembre). *Believable Caustics Reflections.* Descargado de `https://www.alanzucconi.com/2019/09/13/believable-caustics-reflections/`

Lista de Acrónimos y Abreviaturas

2D	de dos dimensiones.
3D	de tres dimensiones.
API	interfaz de programación de aplicaciones.
BRDF	Bidirectional Reflectance Distribution Function.
CPU	Central Processing Unit.
FPS	Frames Per Second.
GLSL	OpenGL Shading Language.
GPU	Graphics Processing Unit.
LoD	Level of Detail.
MVS	Microsoft Visual Studio.
RGB	Red, Green and Blue.
TFG	trabajo fin de grado.

A. Anexo 1 - Obtención de la matriz modelo o de transformación

La **matriz identidad** es la matriz base de la que parte para obtener el resto, si se aplica sobre un elemento este no modifica sus propiedades. Sería el equivalente a realizar una multiplicación por uno.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 1 \cdot 1 \\ 1 \cdot 2 \\ 1 \cdot 3 \\ 1 \cdot 4 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$$

Figura A.1: Matriz identidad
Fuente: de Vries (2020)

La **matriz de translación** se obtiene aplicando el vector que se desean desplazar los elementos a los valores de los ejes. Para ello se situan en la cuarta columna, si la matriz hubiese sido de tamaño 3x3 no se habría podido realizar esta operación, este es uno de los motivos por los que se utilizan matrices de tamaño 4x4.

$$\begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x + T_x \\ y + T_y \\ z + T_z \\ 1 \end{pmatrix}$$

Figura A.2: Matriz de translación
Fuente: de Vries (2020)

La **matriz de rotación** es bastante más compleja que la anterior como se puede ver en la figura A.3, teniendo una matriz por cada eje aplicando senos y cosenos en posiciones distintas de la matriz. Se van a combinar estas tres, lo que causará un problema conocido como bloqueo del cardán (del inglés *Gimbal lock*) por el que no podremos rotar en uno de los ejes perdiendo un grado de libertad.

Los quaterniones es un sistema matemático que utiliza números complejos con el que podermos resolver este problema, la librería GLM incropora funciones que facilitan hacer uso

de ellos.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ \cos\theta \cdot y - \sin\theta \cdot z \\ \sin\theta \cdot y + \cos\theta \cdot z \\ 1 \end{pmatrix}$$

$$\begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\theta \cdot x + \sin\theta \cdot z \\ y \\ -\sin\theta \cdot x + \cos\theta \cdot z \\ 1 \end{pmatrix}$$

$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\theta \cdot x - \sin\theta \cdot y \\ \sin\theta \cdot x + \cos\theta \cdot y \\ z \\ 1 \end{pmatrix}$$

(a) Sin combinar

$$\begin{bmatrix} \cos\theta + {R_x}^2(1-\cos\theta) & R_xR_y(1-\cos\theta) - R_z\sin\theta & R_xR_z(1-\cos\theta) + R_y\sin\theta & 0 \\ R_yR_x(1-\cos\theta) + R_z\sin\theta & \cos\theta + {R_y}^2(1-\cos\theta) & R_yR_z(1-\cos\theta) - R_x\sin\theta & 0 \\ R_zR_x(1-\cos\theta) - R_y\sin\theta & R_zR_y(1-\cos\theta) + R_x\sin\theta & \cos\theta + {R_z}^2(1-\cos\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

(b) Combinadas

Figura A.3: Matrices de rotación
Fuente: de Vries (2020)

La **matriz de escalado** se obtiene multiplicando los números por los que se quiere incrementar el tamaño en la diagonal de la matriz identidad, figura A.4. De forma que, como resultado, se multiplicará por los valores de cada eje.

$$\begin{bmatrix} S_1 & 0 & 0 & 0 \\ 0 & S_2 & 0 & 0 \\ 0 & 0 & S_3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} S_1 \cdot x \\ S_2 \cdot y \\ S_3 \cdot z \\ 1 \end{pmatrix}$$

Figura A.4: Matriz de escalado
Fuente: de Vries (2020)

Por último, para obtener la **matriz de transformación** se combinan estas tres matrices multiplicando primero la matriz de translación, luego la de rotación y luego la de escalado.

B. Anexo 2 - Obtención de las matrices de vista y proyección

La **matriz de vista** (del inglés *view matrix*) aplicada sobre los objetos de la escena, que se encuentran en coordenadas de mundo, los posiciona respecto a la cámara como origen. Para obtener esta matriz es necesario disponer los vectores de su posición, dirección a la que apunta, el vector en la dirección positiva del eje X y el que apunta en la positiva del eje Y. Estas direcciones se deben a que OpenGL interpreta el eje X positivo como el que apunta hacia la derecha y el eje Y positivo como el que apunta hacia arriba.

En la figura B.1 se puede ver como realizar el cálculo con los datos previamente obtenidos para obtener la matriz View. Donde R es el vector de la derecha, U es el vector que apunta hacia arriba, D es el vector de la dirección y P es el vector de la posición de la cámara.

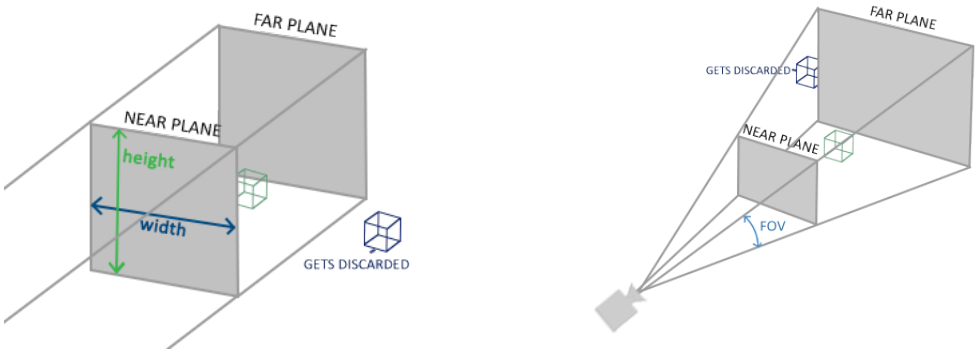
$$LookAt = \begin{bmatrix} R_x & R_y & R_z & 0 \\ U_x & U_y & U_z & 0 \\ D_x & D_y & D_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & -P_x \\ 0 & 1 & 0 & -P_y \\ 0 & 0 & 1 & -P_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figura B.1: Cálculo para obtener la matriz de vista
Fuente: de Vries (2020)

La **matriz de proyección** (del inglés *proyection matrix*) aplicada a los elementos en coordenadas de vista los proyecta a un plano 2D que representa la pantalla donde se dibujan los elementos. Transformando la anchura y altura máxima de los elementos posicionados en la escena a una coordenada definida entre -1.0 y 1.0. En caso de que el objeto se encuentre fuera del rango del cámara definido se descartará su dibujado, es lo que se conoce como ***Clipping***. Hay dos tipos de proyecciones:

- La **ortográfica**, como podemos ver en la figura B.2a, no tiene en cuenta la distancia y se captura como si fuese un cubo que se extiende. Tiene en cuenta los valores de posición más próximo y lejano, así como, donde comienza y finaliza el plano de la cámara.
- La **perspectiva** se basa en el efecto que se produce en la realidad disminuyendo los objetos con la distancia como se ve en la figura B.2b. Aparte de los valores anteriores se tiene en cuenta el campo de visión de la cámara.

La librería GLM proporciona los métodos para calcular la matriz de proyección dependiendo de cuál de las dos proyecciones se quiera aplicar.



(a) Ortográfica **(b)** Perspectiva

Figura B.2: Tipos de proyección

Código B.1: Ejemplo en C++ para calcular la matriz de proyección ortogonal

```
float left = 0.0;
float rigt = 800.0;
float bottom = 0.0;
float top = 600.0;
float near = 0.1;
float far = 100.0;
projection = glm::ortho(left, right, bottom, top, near, far);
```

Código B.2: Ejemplo en C++ para calcular la matriz de proyección perspectiva

```
float fov = 45.0;
float ratio = anchuraPantalla / alturaPantalla;
float near = 0.1;
float far = 100.0;
projection = glm::perspective(glm::radians(fov), ratio, near, far);
```

C. Anexo 3 - Código para calcular las ondas compuestas en el *vertex shader*

Código C.1: Código en *GLSL* para calcular las ondas compuestas

```
#version 450 core

layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;
layout (location = 2) in vec3 aTexCoord;

out vec2 TexCoords;
out vec3 Normal;
out vec3 FragPos;

uniform mat4 transform;
uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;
uniform mat4 MVP;
uniform mat4 lightSpaceMatrix;

// waves
uniform float timeElapsed; // Time
uniform int numWaves; // Number of waves
uniform float waveLengths[6]; // Wavelength
uniform float waveSpeeds[6]; // Speed of waves
uniform float waveAmplitudes[6]; // Amplitude of waves
uniform vec2 waveDirections[6]; // Direction of propagation
const float pi = 3.14159;
const float twoPi = 6.28318;


////////////////////////////////////////////////////////////
// APMLITUD ------------------
float numOndaK(int i) {
    float freq = twoPi / waveLengths[i];
    return freq;
}

float getAngle(int i, float x, float z) {
    float ang = dot(vec2(x, z), vec2(waveDirections[i].x, waveDirections[i].y))↩
        ↪ ;
    return ang;
```

```
}

float wave(int i, float x, float z) {
    return waveAmplitudes[i] * sin(numOndaK(i) * (getAngle(i, x, z) + ↩
        ↪ timeElapsed * waveSpeeds[i]));
}
// ----------------------------

// NORMAL --------------------
float DxWave(int i, float x, float z) {
    float k = numOndaK(i);
    float A = waveAmplitudes[i] * waveDirections[i].x * k;
    return A * cos(k * (getAngle(i, x, z) + timeElapsed * waveSpeeds[i]));
}

float DzWave(int i, float x, float z) {
    float k = numOndaK(i);
    float A = waveAmplitudes[i] * waveDirections[i].y * k;
    return A * cos(k * (getAngle(i, x, z) + timeElapsed * waveSpeeds[i]));
}

float wavePosY(float posX, float posZ) {
    float h = 0.0;
    for (int i = 0; i < numWaves; ++i)
        h += wave(i, posX, posZ);
    h = h / float(numWaves);
    return h;
}

vec3 calculateNormal(float posX, float posZ) {
    float derivativeX = 0.0;
    float derivativeZ = 0.0;
    for (int i = 0; i < numWaves; ++i) {
        derivativeX += DxWave(i, posX, posZ);
        derivativeZ += DzWave(i, posX, posZ);
    }
    derivativeX = derivativeX / float(numWaves);
    derivativeZ = derivativeZ / float(numWaves);
    vec3 norm = vec3(-derivativeX, 1.0, -derivativeZ);

    return normalize(norm);
}


// main
void main()
{
    vec3 FinalPos = aPos;
    FinalPos.y = wavePosY(aPos.x, aPos.z);
    vec3 normalNew = calculateNormal(aPos.x, aPos.z);

```

```
89    gl_Position = vec4(FinalPos, 1.0);;
90    FragPos = vec3(model * vec4(FinalPos,1.0));
91
92    Normal = mat3(transpose(inverse(model))) * normalNew;
93    TexCoords = vec2(aTexCoord.xy);
94}
```

D. Anexo 4 - Código para calcular Gerstner en el *vertex shader*

En este anexo se muestran las funciones modificadas del anexo C para aplicar el algoritmo de Gerstner.

Código D.1: Código en *GLSL* para calcular Gerstner

```
uniform float Qdisplacements[6]; // Displacement

// Amplitud --------------------
float WaveSpeed(int i) {
    return -sqrt( (float(gravity) * float(waveLengths[i])) / twoPi );
}

float WavePosX(int i, float x, float z) {
    return Qdisplacements[i] * waveAmplitudes[i] * waveDirections[i].x * cos(↩
        ↪ numOndaK(i) * (getAngle(i, x, z) + timeElapsed * WaveSpeed(i)));
}

float WavePosZ(int i, float x, float z) {
    return Qdisplacements[i] * waveAmplitudes[i] * waveDirections[i].y * cos(↩
        ↪ numOndaK(i) * (getAngle(i, x, z) + timeElapsed * WaveSpeed(i)));
}

// NORMAL --------------------
float DyWave(int i, float x, float z) {
    float k = numOndaK(i);
    float A = waveAmplitudes[i] * Qdisplacements[i] * k;
    return A * sin(k * (getAngle(i, x, z) + timeElapsed * WaveSpeed(i)));
}

vec3 calculatePositions(float posX, float posZ) {
    float offsetX = 0.0;
    float posY = 0.0;
    float offsetZ = 0.0;
    for (int i = 0; i < numWaves; ++i) {
        offsetX += WavePosX(i, posX, posZ);
        posY += WavePosY(i, posX, posZ);
        offsetZ += WavePosZ(i, posX, posZ);
    }
    offsetX = offsetX / float(numWaves);
    posY = posY / float(numWaves);
    offsetZ = offsetZ / float(numWaves);

```

```
36    return vec3(posX + offsetX, posY, posZ + offsetZ);
37}
38
39vec3 calculateNormals(float posX, float posZ) {
40    float derivativeX = 0.0;
41    float derivativeY = 0.0;
42    float derivativeZ = 0.0;
43    for (int i = 0; i < numWaves; ++i) {
44        derivativeX += DxWave(i, posX, posZ);
45        derivativeY += DyWave(i, posX, posZ);
46        derivativeZ += DzWave(i, posX, posZ);
47    }
48
49    derivativeX = derivativeX / float(numWaves);
50    derivativeY = derivativeY / float(numWaves);
51    derivativeZ = derivativeZ / float(numWaves);
52    vec3 norm = vec3(-derivativeX, 1-derivativeY, -derivativeZ);
53
54    return normalize(norm);
55}
```

E. Anexo 5 - Gráficas del coste en milisegundos de los algoritmos de oleaje y las técnicas de iluminación

A continuación, se muestran las gráficas presentadas en el apartado 6 pero con el coste en milisegundos:

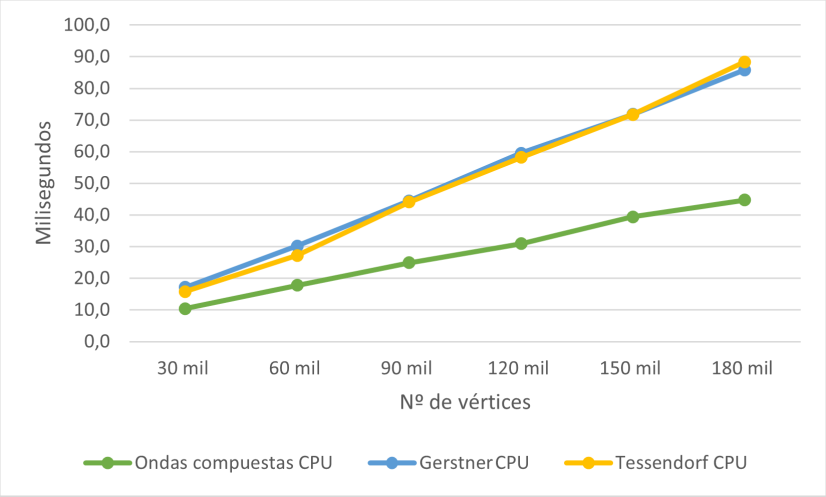
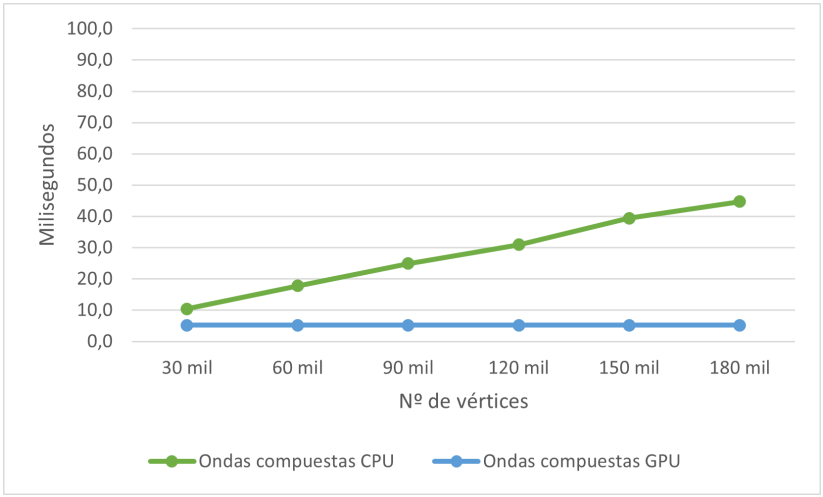
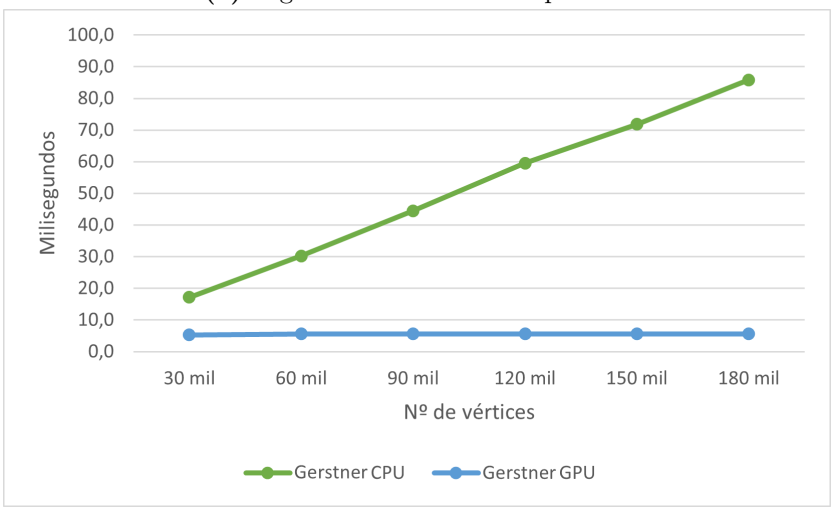


Figura E.1: Gráfica que muestra el rendimiento en milisegundos al aplicar los algoritmos de oleaje en la *CPU*



(a) Algoritmo de Ondas compuestas



(b) Algoritmo de Gerstner

Figura E.2: Gráficas que muestran la comparativa en milisegundos de implementar los algoritmos de la *CPU* en la *GPU*

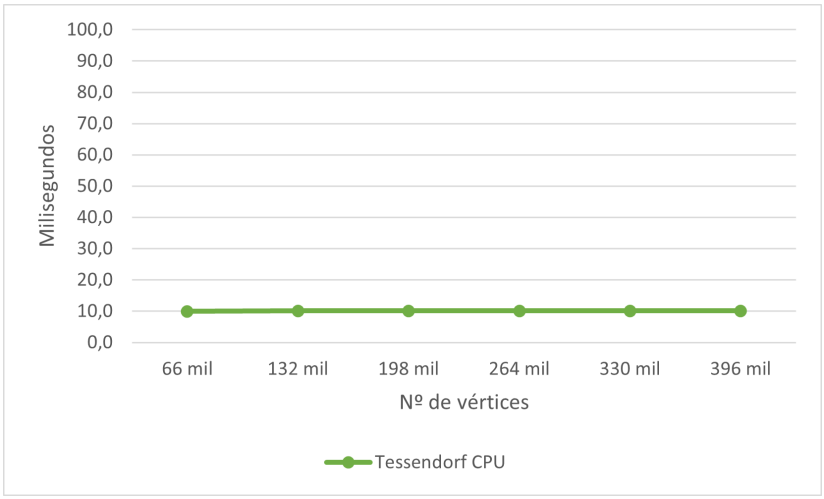


Figura E.3: Gráfica que muestra el rendimiento en milisegundos de Tessendorf al solo aplicar el cálculo del algoritmo a una sección de la malla

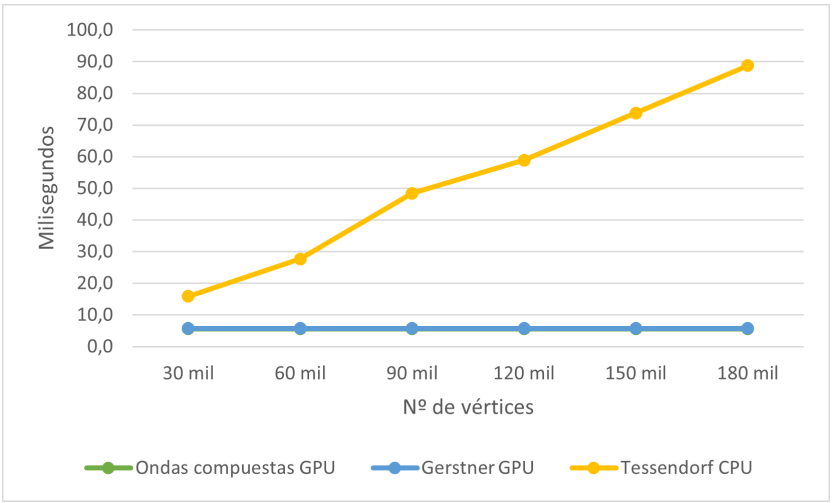


Figura E.4: Gráfica que muestra el coste final en milisegundos de los algoritmos junto a las técnicas de iluminación